New Jersey Institute of Technology
Digital Commons @ NJIT

Theses | Electronic Theses and Dissertations

8-31-1990

Decomposing non-product form queuing lattices through genetic algorithm

Wei Han
New Jersey Institute of Technology

Follow this and additional works at: https://digitalcommons.njit.edu/theses

 Part of the Electrical and Electronics Commons

Recommended Citation

Han, Wei, "Decomposing non-product form queuing lattices through genetic algorithm" (1990). *Theses*. 2711.
https://digitalcommons.njit.edu/theses/2711

This Thesis is brought to you for free and open access by the Electronic Theses and Dissertations at Digital Commons @ NJIT. It has been accepted for inclusion in Theses by an authorized administrator of Digital Commons @ NJIT. For more information, please contact digitalcommons@njit.edu.

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT New Jersey's Science & Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

Decomposing the Non-Product Queueing Lattice Through Genetic Algorithm

Electrical and Computer Engineering Department

Wei Han

Abstract

In this paper, a Genetic Algorithm technique is adapted to decompose the state transition lattice of a class of non-product form queueing models. Genetic Algorithms are search algorithms based upon the mechanics of natural genetics. They combine a survival-of-the-fittest among string structures with a structured, yet randomized, information exchange to form a search algorithm with some of the innovative flair of human search. While randomized, genetic algorithms are no simple random walk. They efficiently exploit historical information to speculate on new search points with inproved performance. Here genetic algorithms is applied to a non-product queueing lattices optimization problem. Only the lattice of type A structure are considered. By applying this technique, the lattice is decomposed into solvable subsets which can be solved sequentially and independently.

Decomposing Non-Product Form Queueing Lattices Through Genetic Algorithm

by

Wei Han

Thesis submitted to the faculty of the Graduate School of
the New Jersey Institute of Technology in partial fulfillment of
the requirements for the degree of .
Master of Science in Electrical Engineering
1990

APPROVAL OF THESIS

Title of Thesis: DECOMPOSING NON-PRODUCT FORM QUEUEING LATTICES THROUGH GENETIC ALGORITHM

Name of Candidate: Wei Han

Master of Science in Electrical Engineering

Thesis and Abstract approved by:

9/5/90

Dr. Irving Wang — Date

Assistant Professor

Department of Electrical and Computer Engineering.

New Jersey Institute of Technology.

9-5-90

Dr. Edwin S. H. Hou — Date

Assistant Professor

Department of Electrical and Computer Engineering.

New Jersey Institute of Technology.

9/5/90

Dr. M. C. Zhou — Date

Assistant Professor

Department of Electrical and Computer Engineering.

New Jersey Institute of Technology.

VITA

Name: Wei Han

Address:

Degree and date to be conferred: M. S. E. E., 1990

Date of birth:

Place of birth:

Collegiate Institute	Attended Date	Degree	Date of Degree
Shanghai Univ., China	9/84-7/88	B.S.	July 1988
N. J. I. T., Newark, USA	9/88-9/90	M. S. E. E.	Oct. 1990

ACKNOWLEDGEMENTS

The author would like to thank **Dr. Irving Y. Wang** for his advice and support; **Mr. Cheng-Kuo Hong** who designed and built the basic structure of state transition lattices.

Finally, I wish to give my loving appreciation for the encouragement and support given to me by my parents, and Mr. Chao-Hsing Tan, for all that they have done for me.

Contents

List of Figures

Chapter 1

Introduction

1.1 Product and Non-Product Form

In studying the behavior of computer systems, queueing network models are a useful tool for evaluating performance and studying the interaction of resources, software and workload[1]. The product form solution of the balance equations, when it exists, plays an important role in analyzing such queueing network models. Product form solutions are of great interest because the direct numerical solution of balance equations is computationally expensive.

In [2] it is shown that specific structure in the state transition lattice leads to a sequential method of solution. The method is sequential either in terms of individual state or in terms of groups of states. In both cases, substantial computational savings are possible. A great deal of research has been performed on the subject of product form solutions. Originally, Jackson described the equilibrium state probabilities for open networks of queues with a single class of jobs, Poisson arrival statistics, and exponential service times[3]. These probabilities were of a characteristic "product form":

$$p(n_1, n_2, \cdots, n_m = p_1(n_1)p_2(n_2)\cdots p_m(n_m) \tag{1.1}$$

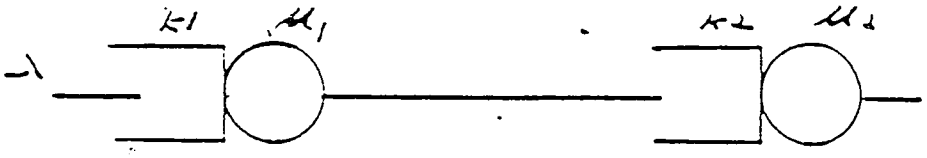
Later, Gordon and Newell analyzed closed queueing networks with each station having an exponential service time distribution[7]. The equilibrium states probability as :

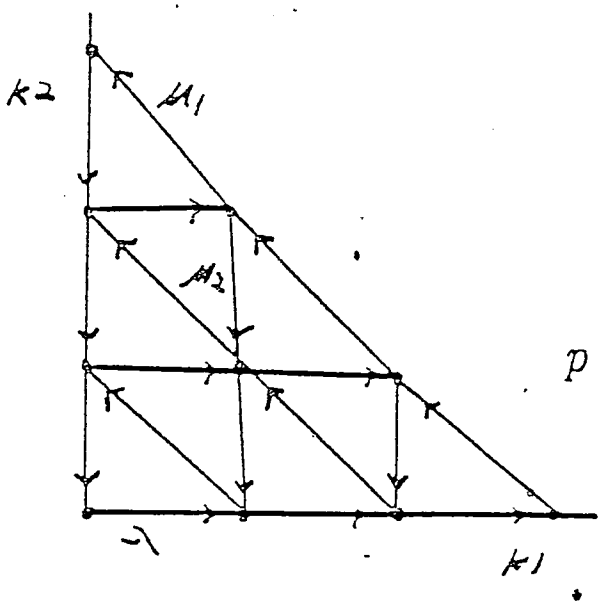
$$p(n_1, n_2, \cdots, n_m) = \frac{1}{G(N)} \prod_{i=1}^{m} f_i(n_i) \tag{1.2}$$

where *G(N)* is a normalization constant chosen to make all the feasible state probabilities sum to one. N is the total number of jobs. The f_i are akin to the marginal probabilities of (1.1).

Product form solutions are of great interest because the direct numerical solution of balance equations is computationally expensive. While the calculation of *G(N)* may not be trivial [5], the evaluation of product form solutions is still computationally and analytically preferable to solving to very large systems of linear equations. In [2] it was shown that the existence of the product form solution corresponds to a decomposition of the state transition lattice (complex) into elementary geometric building blocks (cells).

There is a similarity between the balance equation of queueing network state transition lattices and current conservation equations of resistant circuits. Naturally, the flow in the former involves probability flux[9] rather than current. Transition rates are akin to conductance and equilibrium probabilities are akin to voltages. There are three significant differences though. There is a scaling of equilibrium probabilities to unity which induces probability flux flow in place of voltage sources. The flow direction are predetermined from the transition directions. Finally, the transition rates are labeled in a patterned manner from the queueing schematic. The existence of product form solutions has been characterized in terms of certain types of *queueing networks*. That is the algebraic topology of the **state transition lattice**, shown in





Product Form

$$p(k_1,k_2) = \left(\frac{\lambda}{\mu_1}\right)^{k_1} \left(\frac{\lambda}{\mu_2}\right)^{k_2} p(0,0)$$

Cyclic 3 Queue Network

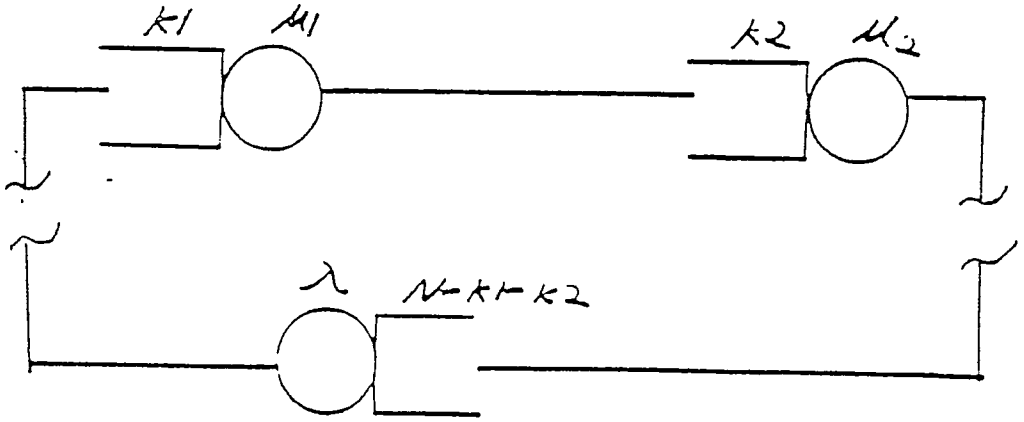


Figure 1.1: Product Form

fig 1.1.

Non-product form queueing networks are far less likely to have a closed form solution for the equilibrium probabilities. It is shown in fig 1.2. Direct solution techniques are prohibitively expensive. Techniques analogous to that of the z-transform can sometimes be used to determine distributions of interest [1,8]. In [1] the sequential decomposition technique was described. Conclusions are given in Chapter 4.

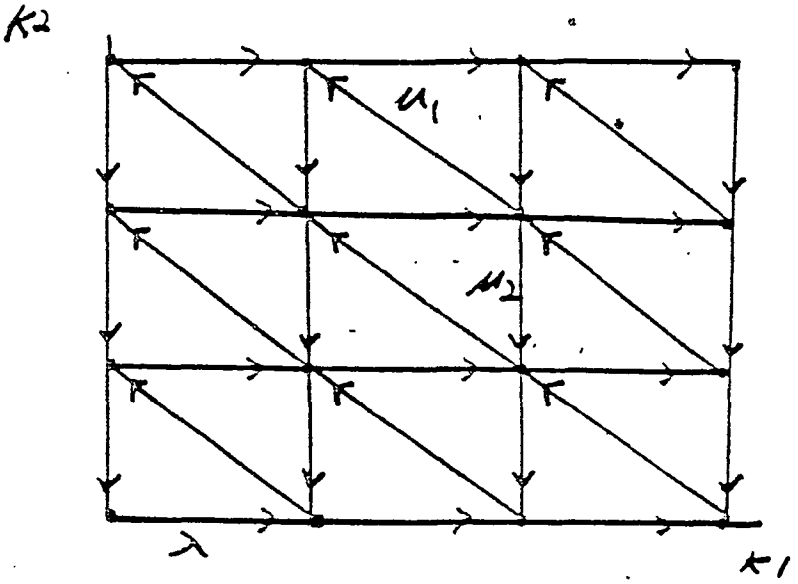
1.2 Sequential Decomposition

This paper deals with the exact computation of the state probabilities of a class of non-product form queueing networks which is of interest for communication and computation system. It is shown that specific structure in the state transition lattice leads to a sequential method of solution. The method is sequential either in terms of individual state or in terms of groups of states.

In [8] a class of non-product form networks is described whose state transition lattices can be shown to be equivalent to a lattice tree of simplexes. In this "flow redirection" method the lattice geometry is manipulated by equivalence transformations. Sequential decomposition refers to the related process of solving one subsets of states at a time for the equilibrium probabilities:

Definition 1.1: a solvable subset of queueing network states is a subset of states for which the equilibrium probabilities can be determined without regarding to the equilibrium probabilities of the remaining unknown states. Here probabilities are determined with respect to a reference probability.

A simple example is presented in Fig 1.3. States S1 and S2 form a solvable subset. The equilibrium probabilities can be determined without regarding to the values associated with the other states. This is done through the following global



Non-Product Form

Figure 1.2: Non-Product Form

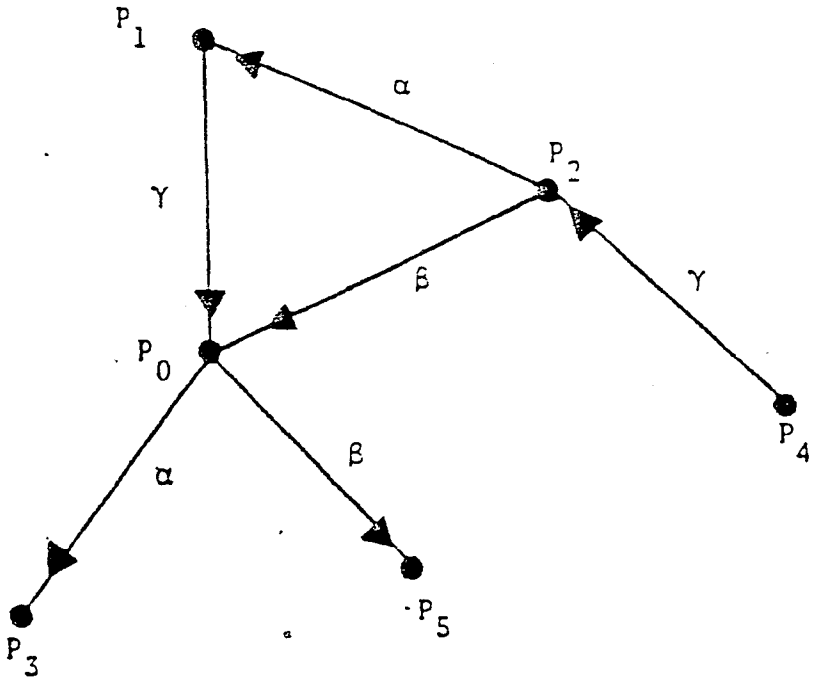


Figure 1.3: A solvable subset

baiance equations:

$$(\alpha + \beta)p_0 = \gamma p_1 + \beta p_2 \tag{1.3}$$

$$\alpha p_2 = \gamma p_1 \tag{1.4}$$

These equations can be solved for the probabilities p_1 and p_2 as a function of reference probability p_0. In fact. a more general principle can be established. The method of sequential decomposition is applicable to queueing systems whose states have the geometry shown in Fig 1.4. Here each circular cluster represents a state or a group of states. Consider the *ith* cluster. the rule is that there must be only one state. with unknown probability. external to the cluster from which a transition(s) entering the cluster originates.

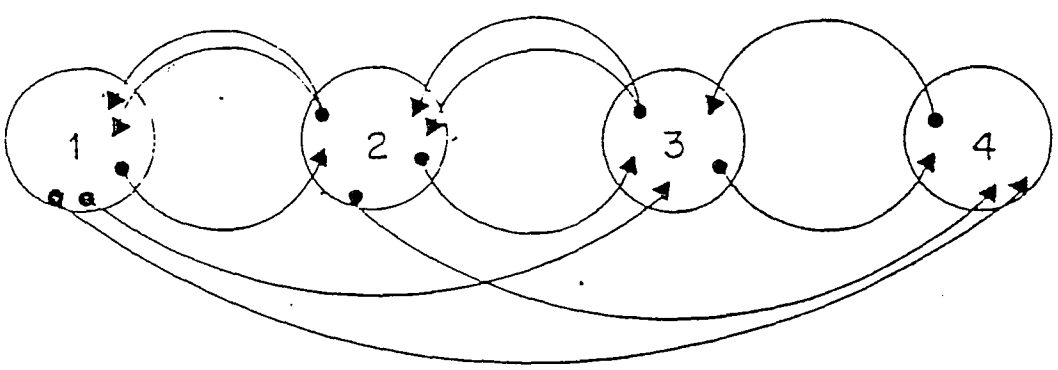


Figure 1.4: The basic structure of Type A

The clusters are solved sequentially, starting from the first cluster to the second and so on. Note that there is no restriction on the number of transitions which may leave the *ith* cluster for destination in the $j = i + 1, i + 2, \cdots$, cluster. The solution equations may not be unique.

Note also that the direct solution of linear equations takes time proportional to the cube of the number of equations. If N states can be solved as M states then the computational effort is proportional to $(N/M)^3 * M$ rather than N^3.

Two types of the structure which allow the state transition lattice to be decomposed into solvable subsets was obtained in [2]. The first type of structure is illustrated in Fig. 1.4 and Fig. 1.6. Here each circular subset represents a state or a group of states. For the ith subset the rule is that there must be only one state. with unknown probability, external to the subset from which a transition(s) entering the subset to the second and so on. There is no restriction on the number of transitions which may

leave the ith subset for destinations in the $j = i+1, i+2, \cdots$ subsets. This type of structure is type A structure.

The second type of structure is illustrated in Fig. 1.5. Here the first subset consists of a single state. The remaining subsets each consist of a state or a group of states. These are arranged in a tree type of configuration with the flow between subsets from the top of the diagram to the bottom and a return flow from the bottom level back to the top level. The subsets may be solved from the top to the bottom. Transitions may traverse several levels as long as the direction of flow is downward. This type referred to as Type B structure.

In this paper, only Type A structure is considered to be decomposed a state transition lattice into a number of solvable subsets, each of which can be solved independently, by using Genetic Algorithm. The state transition of Type A is illustrated in Fig. 1.6.

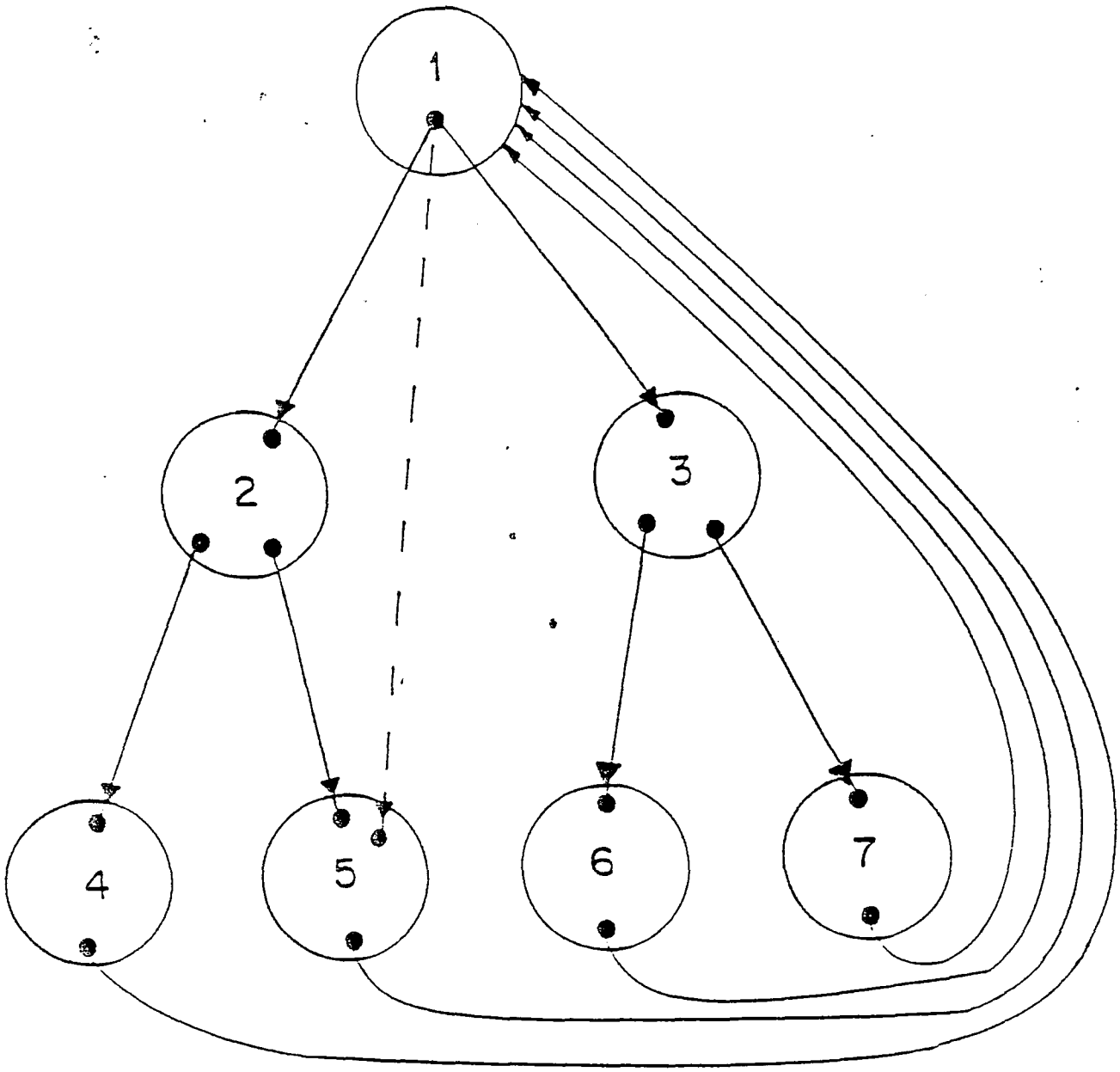


Figure 1.5: The basic structure of Type B

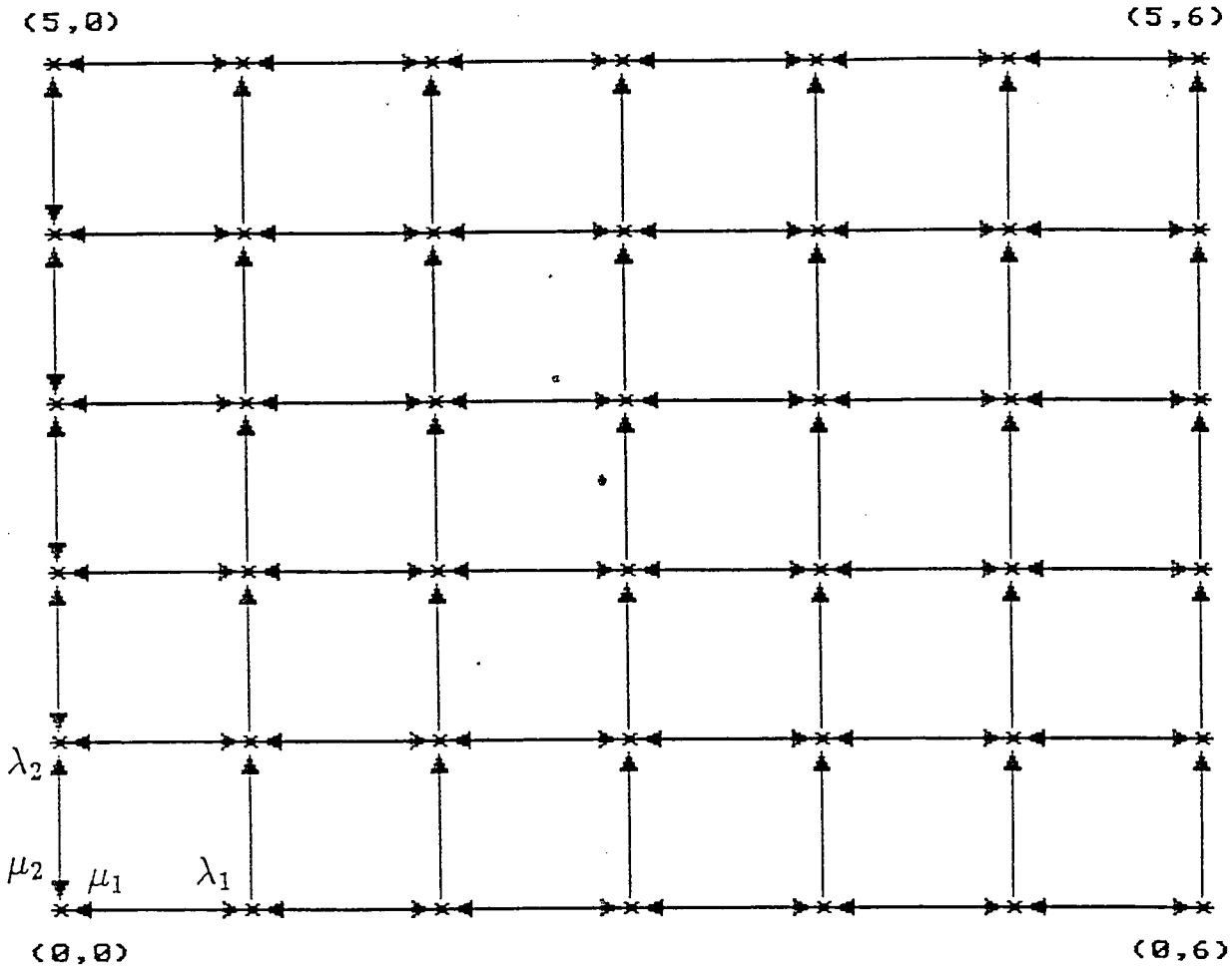


Figure 1.6: A typical example of Type A lattice

Chapter 2

Genetic Algorithm

2.1 Introduction

Upon previous discussion, the task becomes an optimization problem of objective function. While the optimization problem itself is unremarkable (a straightforward parameter optimization problem which has been solved by other methods), the genetic algorithm approach we adopted is noteworthy because it draws from the most successful and longest lived search algorithm. Further more, the GA approach is provably efficient in its exploitation of important similarities, and thus connects to our own notions of innovatative or creative search.

Genetic Algorithm, developed by John Holland, his colleagues and his students at the University of Michigan, provides robust and efficient search in complex spaces. Actually, Genetic Algorithms are search algorithms based upon the mechanics of natural genetics. Survival of the fittest among string structures and structured yet randomized information exchanges are the basic philosophies behind these algorithm. While randomized, genetic algorithm are no simple random walk. It is search procedure that uses random choice as a tool to guide a highly exploitative search through a coding of a parameter space. They have been applied to a variety of function optimization problems, and also, combinational problems such as the Travelling Salesman

Problem.

GA are different from the normal search methods encountered in engineering optimization in the following ways:

1. GA work with a coding of the parameter set not the parameters themselves.

2. GA search from a population of points.

3. GA use probabilistic not deterministic transition rules.

Genetic algorithm requires the natural parameter set of optimization problem to be coded as a finite length string. A variety of coding schemes can and have been used successfully. Because GAs work directly with the underlying code they are not dependent upon continuity of the parameter space and derivative existence.

In many optimization methods, we move gingerly from a single point in the decision space to the next using some decision rule to tell us how to get to the next point. This point-by-point method is dangerous because it often locates false peaks in multimodel search spaces. GAs work from a database of points simultaneously (a population of strings) climbing many peaks in parallel, thus reducing the probability of finding a false peak.

Unlike many methods, GAs use probabilistic decision rules to guide their search. The use of probability does not suggest that the method is simply a random search, however. Genetic algorithms are quite rapid in locating improved performance.

A simple Genetic Algorithm consists of three operators:

- Reproduction
- Crossover
- Mutation

With our simple genetic algorithm we view reproduction as a process by which individual strings are copied according to their fitness(fitness is defined as the non-negative figure of merit we are minimizing, thus, the fitness in genetic algorithm work corresponds to the objective function in normal optimization work). Highly fit strings received higher numbers of copies in the mating pool. There are many ways to do this; we simply give a proportionately higher probability of reproduction to those strings with higher fitness (objective function value). Reproduction is thus the survival-of-the-fittest or emphasis step of the genetic algorithm. The better strings make more copies for mating than the worse.

The rapid convergence usually occurs when super individuals appear in the population. These super individuals will be rewarded with a large number of offspring in the next generation. Since the population size is typically kept constant, the number of offspring allocated to a super individual will prevent some other individuals from contributing any offspring to next generation. These super individuals could be the sub-optimal solutions for the problems, but when they dominate the reproduction process, the search could no longer progress to reach the real optimal solution. Various methods have been proposed to control rapid convergence. One way to control rapid convergence is to control the range of trials allocated to any single individual, so that no individual receives many offspring. The ranking system is one such alternative selection algorithm. In this system, each individual receives an expected number of offspring which is based on the rank of its performance and not on the magnitude. In this paper, the ranking system where the amount of offspring reproduced depends on its ranking of performance and not on the magnitude is adopted.

After reproduction, simple crossover may process in two steps. First, members of the newly reproduced strings in the mating pool are mated at random. Second, each

pair of strings randomly select two positions in each string and swap all the characters between these two positions to create two new strings. More sophisticated crossover techniques are proposed in [10][11] for individual problems. The crossover similar to the one developed in [10] is used in this paper. The process of reproduction and crossover in a genetic algorithm is in this kind of exchange. High-performance notions are repeatedly tested and exchanged in the search for better and better performance.

Mutation is the occasional random alteration of the value of a string position. It is not frequently used in either the artificial or the natural genetic systems, thus it is not adopted in this paper.

So, our implementation of genetic algorithm is as follow:

- Ranking
- Reproduction
- Crossover

2.2 Model Description

In previous sections, it is shown that Genetic Algorithms are mechanically quite simple, involving nothing more than random number generation, string copies, and partial string exchanges. However, together with the reproduction and the structure randomized, information exchange of crossover give genetic algorithms much of their power.

In the algorithm, a population of n strings is numerical coded so that each of them is a complete IDEA or a prescription for performing a particular task. The population contains not just a sample of n IDEAS, rather it contains a multitude of NOTIONS and Ranking of these NOTIONS for task performance. Genetic Algorithms carefully

exploit this wealth of information about important NOTIONS by

- reproducing quality NOTIONS according to their performance
- crossing these NOTIONs with many other high performance NOTIONS from other strings.

The format of the string in our decomposition problem is defined as follow:

$a_1, a_2, a_3, ! \cdots ! \cdots a_M$

Here, a_i is the state id, M is the number of states in the lattice and ! marks the subset boundary. In Fig. 2.1, as an example, the state transition lattice consists of nine states which are grouped into three subsets with the dash lines marking the boundary. the corresponding string can be represented as

2 4 5! 0 1 3 6! 7 8

The ranking of a certain string depends on the number of states in each subset and the number of transitions between subsets in its corresponding lattice. Thus the fitness or the object function of the string is defined as following:

$$objective function = \sum_{i}^{n} T_i + \sum_{i}^{n} M * n_i^3 \tag{2.1}$$

where M is the number of subsets in the string. The smaller value of fitness a string has, the higher rank it is granted. If there is a tie on the values of F's of different strings, then the size of subsets will assume the role of tiebreaker in the ranking determination. Since the computation time for solving a set of linear equations is proportional to the cube of the number of equations in that set, those strings with

lower transitions win higher ranks in the tiebreak. Here n_i is the number of states in the ith subset of the string.

The computation time for solving a set of linear equations is proportional to the cube of the number of equations. This leads to the establishment of the second term of objective function. The rule of constructing a Type A structure yields the first term. Let T_i denote the number of transitions, originating from the states with unknown probabilities and external to the ith subset, going into the ith subset and m denote the number of subsets in one particular decomposition structure. To emphasize the significance of the structure which satisfies the type A rule, T_i is set to 0 if there is only one incoming transition originating from unknown states external to the ith subset. Once a structure is formed, in other words, a string is generated, every subset is inspected and the number of transitions from all subsets which are positioned after this subset is calculated. This grants T_i with its value. T_i is decreased by 1 if it equals to 1 and remains its value otherwise.

According to the rank of the string, certain number of offsprings are generated for each string on a predetermined basis in the reproduction procedure. The crossover is conducted in a fashion that is shown in the following example:

Parent 1: 2 4 5 8! 0 1! 3! 6 7
Parent 2: 0 1 3! 2 4 5 8! 7 6

Starting from the left end of the strings, copy characters from parent 1 and parent 2 to child 1 and child 2 until the first ! in both parents is met.

Child 1: 2 4 5
Child 2: 0 1 3!

Crossover all the succeeding characters prior to the second ! in both parents.

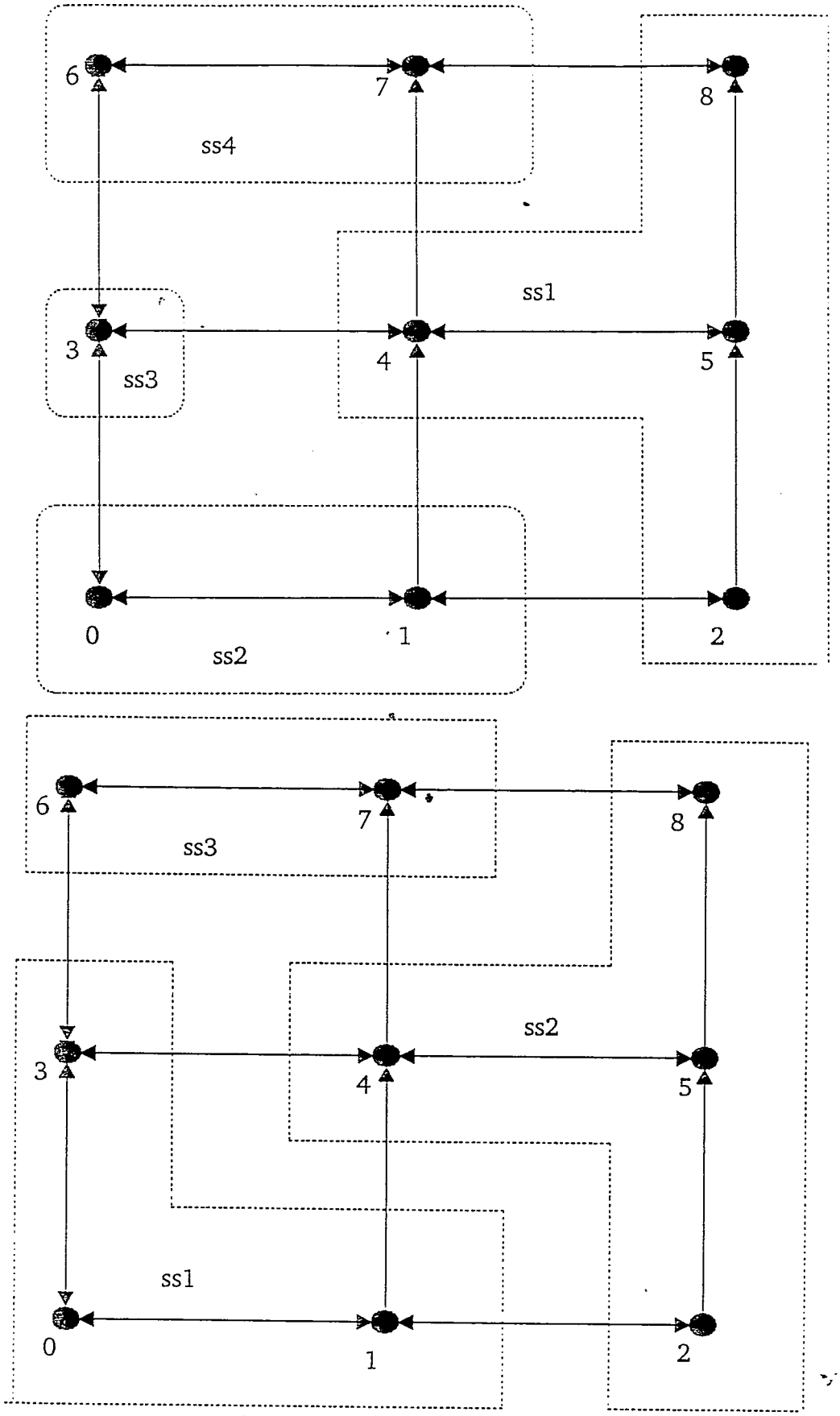


Figure 2.1: Two parents before crossover

Child 1: 2 4 5 2
Child 2: 0 1 3! 8!

Note that the second ! is part of the crossover package. Repeat the above procedure, i.e. alternatively copy and crossover the characters from parents to children, till the right end of the strings is reached. Crossover is an operation in which two "parent" classifiers produce an offspring that is possibly important over both of them. But the crossover points above must be chosen quite carefully, or otherwise the offspring might have no improvement, or even a retrogression. The results are

Child 1: 2 4 5 2 0 1! 6! 6 7
Child 2: 0 1 3! 8! 4 5 3! 7 8

The side-effect of the crossover is that states 2 and 6 appear twice in child 1 but states 3 and 8 are left out. Modifications have to be made to ensure that the legitimacy of the lattice structure is maintained. Thus randomly choose one of the left-outs to replace the second 2 and check

Child 1: 2 4 5 3 0 1! 6! 8 7
Child 2: 0 1 3! 8! 4 5 2! 7 6

if the subsets is legitimate. If not, randomly pick up another left-out for replacement. If none of them are satisfied, separate this state and become a single-state subsets right after the above subset. Similar adjustments are made on the second 6 as well as child 2 and the final products in this example are shown in Fig. 2.2.

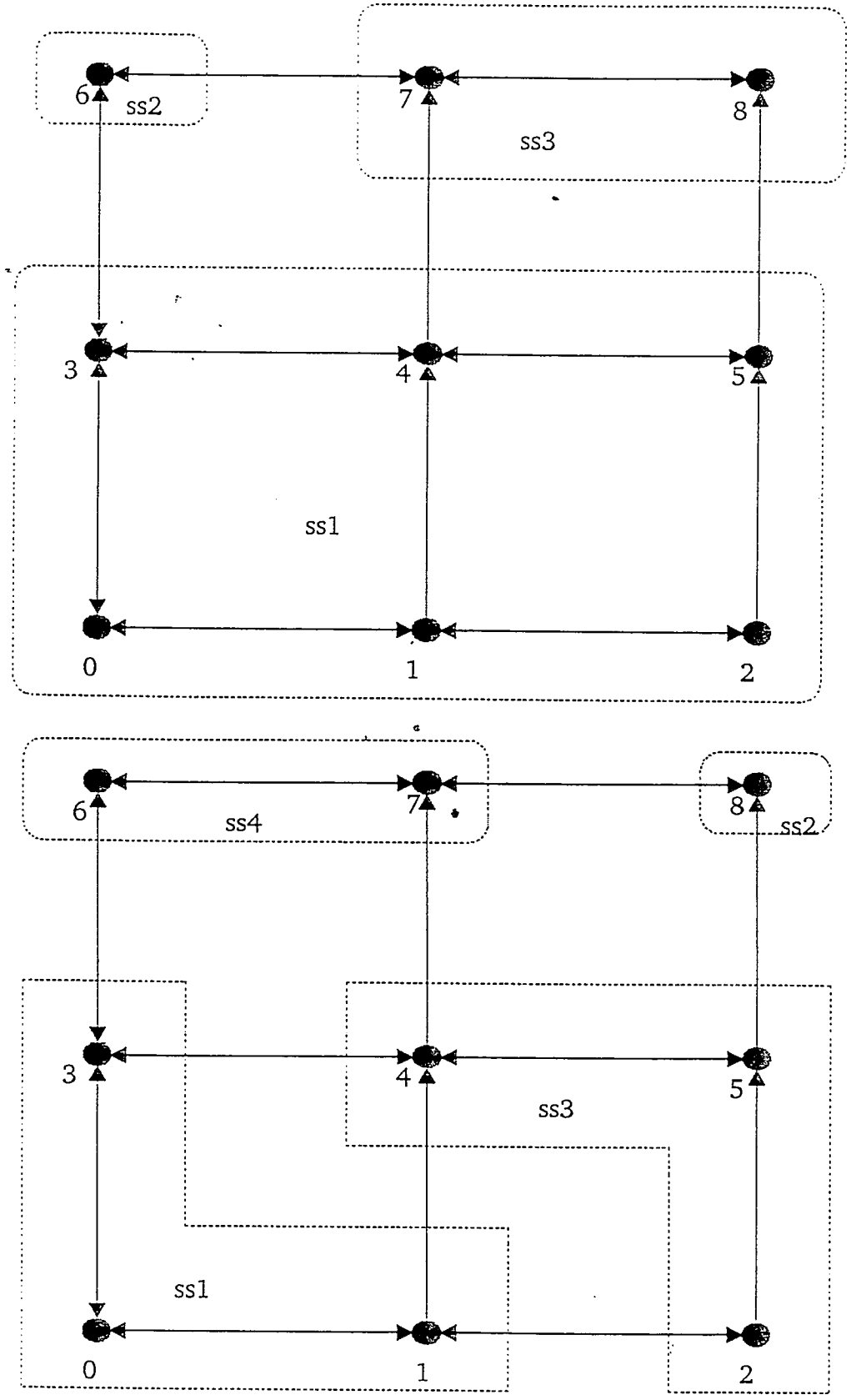


Figure 2.2: Two new children after crossover

Chapter 3

Algorithm

This algorithm is divided into five parts as follows:

Step 1: Initialization.

1. For each state of the transition lattice, the following information is included:
   - the number of its neighbors
   - each neighbor's ID
   - the transition information between the node and its neighbors
2. Divide each string into several subsets randomly.
3. According to the subset lengths obtained above, we start to generate the string. As discussed before, the next state is always picked up from the neighbor pool of current state.
4. Repeat procedures 2,3 100 times to generate 100 strings randomly.

Step 2: Determining the number of transitions.

1. After initialization, we can calculate the number of transitions for each string. Only the number of transitions initiated from the state whose subset is behind

the current subset is considered. Also, if the transition number is the same, we calculate the $\sum M_i^3$. Here M_i is the number of states in the ith subset.

2. Repeat the above step 100 times till all data are found. Each datum carries the information of the number of transition and $\sum M_i^3$.

Step 3: Rank the string according to its associated data.

1. Now we rank the 100 strings by the number of transitions. If there is a tie, we break it by using the quantity of $\sum M_i^3$.

2. If any string has more than pne copies that are generated during initialization, eliminate the extra copies to avoid formation of super individuals.

Step 4: Reproduce.

Pick up the higher ranked strings and duplicate according to their rankings. This number can be adjustable upon the request. Remember that the higher ranking the string stands, the more number of copies it will receive. In my program, I simply selected the best 20 strings to be copied and discarded the rest of them.

Step 5: Crossover.

1. First we randomly pick up two strings from the population as the parents in crossover.

2. After crossover, two child-strings are generated. If either of them contains illegal elements adjust it with the method mentioned above.

Step 6: Simulation

Repeat from step 2 through step 5 40 times.

Step 7: Adjustment

According to the result, adjust subset length and reproduction number in order to obtain the better result.

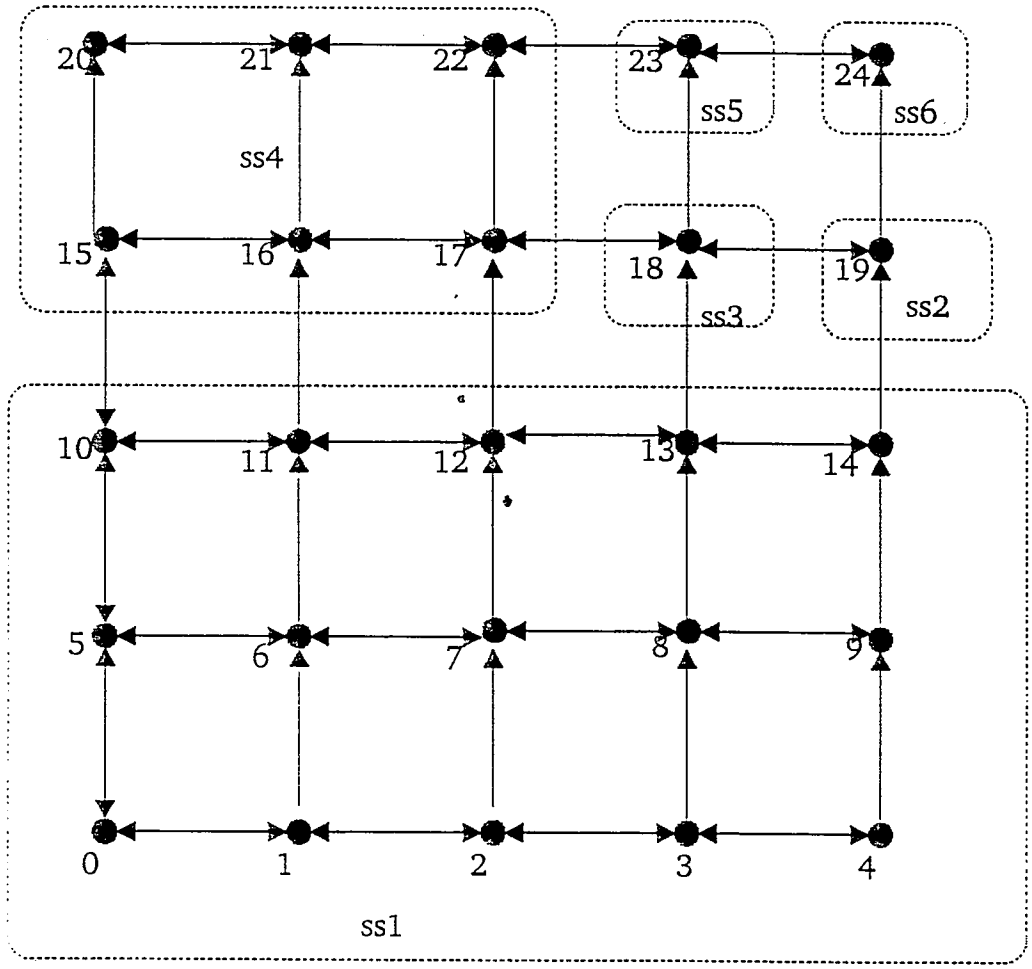


Figure 3.1: The first result of 5*5 structure

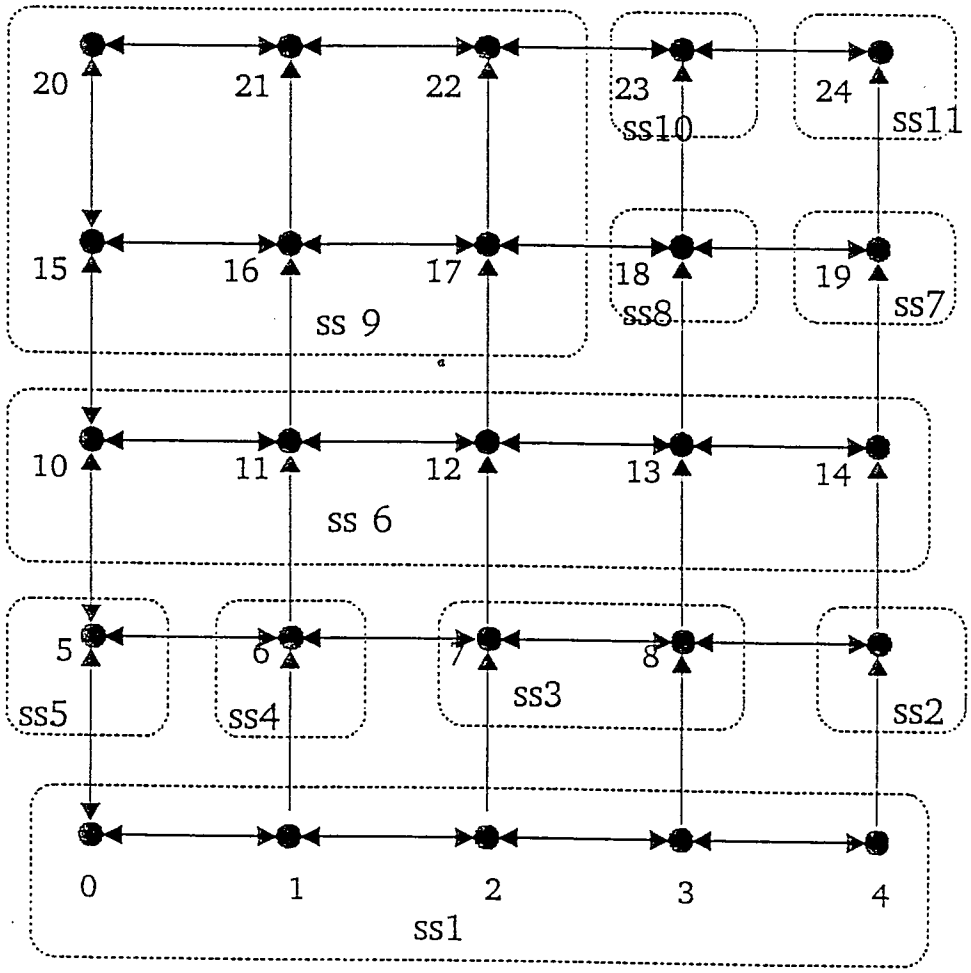


Figure 3.2: Result after adjustment

Chapter 4

Conclusion and Further Suggestion

4.1 Conclusion

Genetic algorithm has been selected to decompose the state transition lattice of Type A strtucture. By dividing the lattice into several solvable subsets which can be solved sequentially, a large amount of computation time can be saved.

4.2 Suggestion

The result from the first round of simulation is not satisfied enough, because the initialized subset length is planned to obtain the zero transition. This results in large subsets, such as in the 5*5 structure, 15 nodes are grouped in one subset. Thus, large amount of computation is still needed. Therefore, I suggest further decomposition of this 15 nodes with algorithm in a second round. The result is obviously better.

Appendix A

List of Program

This program is a simulation program for self-decomposition of algorithm for the state transition lattices of certain non-product queueing protocols.

```
1:  #include <stdio.h> /* 25 NODE strcture version */
2:  #include <math.h>
3:  #define DATA 100
4:  #define DATAF 50
5:
6:    int NODE;
7:    int Tran[DATA][2];              /* array of 50 data's transition */
8:    int min[DATA];
9:    int a[DATA][50];           /* 50 data */
10:   int a1[DATA][50];
11:   int i, j, jind, impo, ll, cn, nxn;
12:   int p, z, cc, variety, seed=1;
13:   int zpop, size, tnp, kin, total, c2;
14:   int flag[DATA];
15:   double drand();
16:   int neno[80];
17:   int tneig[160];
18:   int pop[25];
19:   int B[80];
20:
21:         struct neighbor{
22:               int id;
23:               int status;
24:         };
25:         struct node{
26:               int id;
27:               int no;
28:               struct neighbor nb[25];
29:         } n[25];
30: FILE *fp;
31:
32:
33:
34: main()
35: {
36:   /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
37:    *                                                             *
38:    *       1. Initialize the fifty groups of data                *
39:    *                                                             *
40:    * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
41:
42:
43:
44:    /* 1) initialize the type A with node, neighbor and status */
45:
46:     int nn, mn, im, l, ln, jd, in, x1, y1, jn, ii, jj;
47:     double  drand();
48:     srand(&seed);
49:     variety =DATA;
50:      NODE=25;
51:     for(x1=0; x1<NODE; x1++)
52:        n[x1].no=0;
53:     for(x1=0; x1<80; x1++)
54:    neno[x1]=-1;
55:     i=j=0;
56:           if ((fp=fopen("input25","r"))==NULL) {
57:           printf(" input25 file reading error !\n");
58:           exit(1);
59:    }
60:
61:     for(im=0; im<NODE; im++){
62:         fscanf(fp, "%d", &nn);
63:         mn =nn-n[im].no;
64:         ln = n[im].no-1;
65:         for(l=1; l<=mn; l++){
66:            fscanf(fp, "%d %d", &n[im].nb[ln+l].id, &n[im].nb[ln+l].status);
67:      /*printf("im=%d n[im].id=%d n[im].status=%d\n", im, n[im].nb[ln+l].id, n[im].nb[ln+l].status);*/
68:           jd=n[im].nb[ln+l].id;
69:           jn = -n[im].nb[ln+l].status;
70:           in = n[jd].no;
71:       n[jd].nb[in].id = im;
72:           n[jd].nb[in].status = jn;
73:           n[jd].no++;
74:         }
75:         n[im].no=nn;
76:     }
77: /*
78: for(x1=0; x1<NODE; x1++){
79:    for(y1=0; y1<n[x1].no; y1++)
```

```
80:     printf("x1=%d n[x1].id=%d n[x1].stutas=%d\n", x1,n[x1].nb[y1].id, n[x1].nb[y1].status);
81:   }
82:   */
83:    nxn=0;
84:   for(y1=0; y1<DATA; y1++){
85:      for(x1=NODE; x1<(2*NODE); x1++)
86:         a[y1][x1] = 0;
87:     }
88:   for(y1=0; y1<DATA; y1++){
89:      for(x1=0; x1<NODE; x1++)
90:        { a[y1][x1]=-1; }
91:    }
92:   for(x1=0; x1<DATA; x1++){
93:     for (y1=0; y1<2; y1++)
94:     Tran[x1][y1] = 0;
95:     }
96:   for(x1=0; x1<DATA; x1++){
97:      for(y1=0; y1<2*NODE; y1++)
98:      a1[x1][y1]=0;
99:     }
100:
101:
102:  /* 2) creat subset length for 50 groups of data */
103:        i=j=0;
104:     for(i=0; i<variety; i++){
105:        while (j<=NODE){
106:           j = j+(int) 16*drand();
107:           a[i][NODE-1+j] = 1;           /* if exceed the size? */
108:          }
109:        j=0;
110:     }              /* for */
111:
112:  /*
113:    for(y1=0; y1<variety; y1++){
114:       for(x1=NODE; x1<2*NODE; x1++)
115:     printf("ss length, x1=%d a[y1][x1]=%d\n",x1, a[y1][x1]);
116:     }
117:  */
118:    for(i=0; i<variety; i++){
119:         j=0;
120:      cc=c2=zpop=0;
121:      size = NODE-1;
122:      for(x1=0; x1<NODE; x1++)  pop[x1]=x1;
123:      for(x1=0; x1<80; x1++)  neno[x1]=-1;
124:
125:      a[i][j] = p = crand();
126:      while(j <(NODE-1) )
127:       {
128:          while((a[i][j+NODE] ==0) && (j<(NODE-1)))
129:          {
130:           c2=n[p].no;   /* c2 is total No. of neig */
131:           neig();
132:         /*  printf("return p=%d", p);  */
133:            while( (cc>0) && (repeat()==1) && (j<(NODE-1)))
134:            {
135:          /*  printf("i=%d j=%d a[i][j]=%d\n", i, j, a[i][j]);  */
136:         z=(int) cc*drand();
137:         p=neno[z];
138:            if(z != (cc-1)){     /* cc is the index of array neno */
139:         neno[z] = neno[cc-1];
140:         neno[cc-1]=-1;
141:             }
142:         cc--;
143:        }
144:             if(repeat()!=1 && j<(NODE-1)){
145:            j++;
146:           a[i][j] = p;
147:               /* printf(" check i,i=%d j=%d %d\n", i, j, a[i][j]); */
148:                inkill();
149:             }
150:            else
151:           a[i][NODE+j] = 1;
152:            }  /* while ==0 */
153:
154:            if(j != (NODE-1)){
155:           /* printf("######3\n");  */
156:             j++;
157:             a[i][j]=p=crand();
158:             for(x1=0; x1<80; x1++)  neno[x1] = -1;
```

```
159:             cc=c2=0;
160:             }
161:       }    /* while j<NODE */
162:     }   /* for */
163:    /*
164:     for(ii=0; ii<variety; ii++){
165:       for(jj=0; jj<NODE; jj++)
166:         printf("initial gene,ii=%d jj=%d a[ii][jj]=%d\n", ii, jj, a[ii][jj]);
167:      }
168:    */
169:
170:
171:      for(nxn=0; nxn<2; nxn++){
172:
173:        transition(); /* return array of value indicated transition of 50 group*/
174:                      /* pass a array of transition Tran[50] to the sort function */
175:        sort();             /* return array of 50 data by order */
176:                            /* pass the min[50] which is by order */
177:
178:         kill();            /* kill the gene with same tr and cube n */
179:
180:       if(nxn==1){
181:          for(i=0; i<variety; i++){
182:             for(j=0; j<50; j++)
183:          a1[i][j]=0;
184:               a[i][j]=0;
185:           }
186:          redo();
187:        }
188:
189:       for(x1=0; x1<variety; x1++){
190:         for(y1=0; y1<2; y1++)
191:            Tran[x1][y1]=0;
192:        }
193:
194:        reproduce();         /* reproduce the best data, the most amount */
195:                           /* pass a[100] which is the best first 30 data of a1[50] */
196:    variety=DATAF;
197:        cross();           /* return two new generated legal data */
198:    printf(" A NEW LOOP START variety=%d nxn=%d\n", variety,nxn);
199:       for(x1=0; x1<variety; x1++){
200:          for(y1=0; y1<2*NODE; y1++)
201:               a[x1][y1]=a1[x1][y1];
202:        }
203:     }  /* for */
204:    }  /* main */
205:
206:       /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
207:        *                                                               *
208:        *       Varieties functions                                     *
209:        * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
210:
211:
212:
213:      /* 1) creat random node between 0-8 */
214:
215:      int crand()
216:      {
217:      double  dd;
218:      dd=0;
219:
220:      dd=drand();
221:       zpop =(int) (size+1)*dd;
222:       p = pop[zpop];
223:       inkill();
224:       return(p);
225:      }
226:
227:    /*inkill */
228:    inkill()
229:    {
230:      int x1;
231:      x1=0;
232:
233:       while(p != pop[zpop])
234:         { zpop++;}
235:        if(zpop!=size){
236:         pop[zpop] = pop[size]; }
237:         pop[size] =-1;
```

```
238:        size--;
239:        zpop=0;
240:       return;
241:    }
242:
243:      /* 2) generate random number which less than 1 */
244:
245:      double drand()
246:      {
247:      double x, y;
248:      x = (double) rand();
249:      y = (double) rand();
250:        if ( x==y )
251:          ++y;
252:          x = (x > y)? y/x:x/y;
253:          return( x);
254:     }
255:
256:
257:      /* 3) check if new generated node is repeated previously */
258:
259:      int repeat()
260:      {
261:        int  m1;
262:        m1=0;
263:        ll=0;
264:
265:          while ((m1<=j) && (p != a[i][m1]))
266:          {  m1++;  }
267:            if(p==a[i][m1]) ll=1;
268:            else ll=0;
269:        return(ll);
270:      }
271:
272:    /*  create a neighbor array to be ready to pick up by ss selection */
273:
274:    int neig()
275:    {
276:       int x1, pp;
277:          x1=pp=0;
278:
279:       for(pp=cc; pp<c2+cc; pp++){
280:         neno[pp] = n[p].nb[pp-cc].id;}
281:       z=(int) (cc+c2)*drand();
282:       cc=c2+cc-1;
283:       p=neno[z];
284:          if(z != cc){
285:       neno[z] = neno[cc];
286:       neno[cc]=-1;
287:           }
288:
289:     return(p);
290:    }
291:
292:    /*********************************************************************/
293:    /*  4) caculate the transition */
294:    transition()
295:    {
296:      int ss[25][25];       /* ss is the array of ss No. and ID */
297:      int ssno, x1, y1, d2, tm, sid;
298:      int tk, tp, vary, tng, tran, t1;
299:      for(y1=0; y1<NODE; y1++)  {
300:        for(x1=0; x1<NODE; x1++)
301:        ss[y1][x1] = -1;
302:      }   /* intialize the ss array of the Max size NODE */
303:     if(nxn>=1)  printf("is it did transition\n");
304:    for(i=0; i<variety; i++){
305:        j=0;
306:        for(x1=0; x1<=ssno; x1++){
307:           for(y1=0; y1<=NODE-1; y1++)
308:                ss[x1][y1]=-1;
309:          }   /* reinitialize the ss array before start a new germ */
310:          for(x1=0; x1<=160; x1++)  tneig[x1]=-1;
311:        x1=y1=ssno=sid=0;
312:        while(j<NODE)
313:        {
314:          while(a[i][j+NODE] != 1 && j < NODE)
315:          {
316:       ss[x1][y1]=a[i][j];
```

```
317:      y1++;
318:      j++;
319:           ssno=x1;
320:         }
321:        if(j != NODE){
322:         ss[x1][y1]=a[i][j];
323:         ssno=x1;
324:         sid=sid+(y1+1)*(y1+1)*(y1+1);
325:         y1=0;
326:         x1++;
327:         j++;
328:        }
329:      } /* while j<NODE */   /* finish one gene with ss */
330:      j=NODE-1;
331:      while(a[i][j+NODE] ==0){
332:           j--;}
333:       sid=sid+(NODE-1-j)*(NODE-1-j)*(NODE-1-j);
334:    Tran[i][1]=sid;      /* finish cube of n as 2nd element in array of Tran */
335:
336:
337:       tk=tran=0;   /* start with the frist subset */
338:       while(tk<ssno){  /* end with ssno-1 */
339:        x1=tp=t1=tng=tran=0;
340:        for(y1=0; y1<=160; y1++)  tneig[y1]=-1;
341:         while(ss[tk][x1] != -1){
342:             tp=ss[tk][x1];
343:             t1=n[tp].no;
344:             tm=0;
345:             tneig[tng+80]=tp;    /* store source node ID in last 80 position */
346:            while(tm<t1){
347:                tnp=n[tp].nb[tm].id;
348:                 tneig[tng]=tnp;
349:                 tng++;
350:                 tm++;
351:            }    /* store current neig of node in ss[tk] in tneig[tng] */
352:           x1++;
353:         }
354:
355:          tng=0;
356:       while(tneig[tng]!=-1){
357:            tnp=tneig[tng];
358:           tm=tk+1;
359:         while(tm<=ssno){
360:            x1=0;
361:            vary=79;
362:           while(tnp!=ss[tm][x1] && ss[tm][x1]!=-1)
363:           {        x1++; }
364:           if((tnp==ss[tm][x1]) && (tneig[tng+80]!=-1))
365:              {
366:                tp=tneig[tng+80];
367:           if(n[tp].nb[0].status<=0)
368:                    tran++;
369:             }
370:           else if((tnp==ss[tm][x1]) && (tneig[tng+80]==-1))
371:                 {
372:                      d2=1;
373:                while(tneig[tng+vary]==-1)
374:                   { vary--;
375:                     d2++;
376:                     }
377:               tp=tneig[tng+vary];
378:                   if(n[tp].nb[d2].status<=0)
379:                         tran++;
380:                 }
381:              tm++;
382:           }    /*  while tm<=ssno */
383:             tng++;
384:         }   /* while tng is out of supply */
385:        if(tran==1){
386:           tran=0;
387:         }
388:        else  Tran[i][0]=tran+Tran[i][0];
389:       tk++;
390:     }   /* big while tk<=ssno */
391:    }  /* for loop */
392:
393:     return;
394:    }  /* transition function */
395:    /*************************************************************************/
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* 5) selection sorting */
sort()      /* Tran[q] is for calculate the transition from origial data */
     /* min[tr] is sorted array of transition */
{
   int temp1, temp2, con, tr;
   for(con=0; con<DATA; con++){
       min[con]=0;
       flag[con]=0;
       }
   i=tr=temp1=temp2=0;

      flag[tr] = i;
      min[tr] = Tran[i][0];    /* initial the first element */
       for(i=1; i<variety; i++){
     if(Tran[i][0] < min[tr] || Tran[i][0] == min[tr]){
               con=tr;
   /*      printf("go into if loop tr=%d\n", tr);*/
           while(min[tr] > Tran[i][0] && tr>=0){
          temp1 = min[tr];
          temp2 = flag[tr];
              min[tr] = Tran[i][0];
          flag[tr] = i;
          min[tr+1] = temp1;
          flag[tr+1] = temp2;
     /*printf("great than tr=%d fl=%d min=%d i=%d\n", tr,flag[tr],min[tr],i); */
               tr--;
            }  /* while loop */
           if(Tran[i][0]==min[tr] && tr>=0){
                temp1=flag[tr];
     /*    printf("if equaltr=%d fl=%d min=%d i=%d\n", tr, flag[tr],min[tr],i);*/
           if((Tran[i][1]<Tran[temp1][1] || Tran[i][1]==Tran[temp1][1]) && tr>=0){

                while((Tran[i][0]==Tran[temp1][0]) && (Tran[i][1]<Tran[temp1][1] || Tran[i][1]==Tran[temp1][1]) && tr>
               {
                       temp1 = min[tr];
                   temp2 = flag[tr];
                        min[tr] = Tran[i][0];
                   flag[tr] = i;
                   min[tr+1]=temp1;
                   flag[tr+1] = temp2;
                   temp1=flag[--tr];
            /*       printf("if Tran[i]<Tran[temp1]i=%d", i);*/
           /*   printf("tr=%d flag=%d min=%d\n", tr, flag[tr], min[tr]);*/
                 }  /* while */
             }      /* if loop */
                  else{
                  min[++tr]=Tran[i][0];
                  flag[tr]=i;
            /* printf("keep the same order put into min arrayi=%d", i);*/
           /*printf("tr=%d min[tr]=%d flag[tr]=%d\n", tr, min[tr], flag[tr]);(*/
                }
           }
            tr=con+1;
           }   /* big if loop */
      else{
          flag[++tr] = i;
          min[tr] = Tran[i][0];
       /*printf("less than tr=%d fl=%d min=%d i=%d\n", tr,flag[tr],min[tr],i); */
           }
    /*   printf("&&&&&&&&&&&&&&&&&&&&just before returntr=%d\n", tr);*/
       }               /* for */

  return;
 }                   /* sorting function */


 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
kill()
{
  int yy, y1;

impo=variety-1;

for(i=0; i<impo; i++){
  kin=1;
  while((min[i]==min[i+kin]) && (Tran[flag[i]][1]==Tran[flag[i+kin]][1]) && (i<impo))
  {
```

```
    if(compare()==0){
/*printf("i=%d i+kin=%d f[i]=%d f[i+kin]=%d\n", i,i+kin,flag[i],flag[i+kin]) */

     for(y1=i+kin; y1<impo; y1++){
        min[y1]=min[y1+1];
        flag[y1]=flag[y1+1];
      }
   /* printf("i=%d i+kin=%d impo=%d\n", i, i+kin, impo);*/
   impo--;
     } /* if do compare */
    else kin++;
  } /* while */
} /* for */

 printf("after kill same value,variety=%d impo=%d\n",variety,impo);
/*
if(nxn==0){
for(y1=0; y1<variety; y1++){
 printf("after kill same, min[y1]=%d flag[y1]=%d\n", min[y1],flag[y1]);}
 for(y1=0; y1<variety; y1++){
 for(yy=0; yy<2*NODE; yy++){
  printf("y1=%d yy=%d a[y1][yy]=%d\n", y1, yy,a[y1][yy]); }
 }
} */

y1=flag[0];
printf("y1=%d Tr[y1][0]=%d Tr[y1][1]=%d\n", y1,Tran[y1][0],Tran[y1][1]);
printf("min[0]=%d flag[0]=%d\n", min[0],flag[0]);
  for(yy=0; yy<2*NODE; yy++)
printf("yy=%d a[y1][yy]=%d\n", yy,a[y1][yy]);

 return;
} /* kill function */

/* compare two gene are they the same */
int compare()
{
  int cim, x1, xy, temp, x, cp;
  int com[2][25];

 for(xy=0; xy<NODE; xy++){
    com[0][xy]=-1;
    com[1][xy]=-1;
  }
 x1=flag[i];
/* printf("in compare, x1=%d i=%d\n", x1, i); */
for(xy=0; xy<2; xy++){
  cp=cim=j=temp=0;
  com[xy][0]=a[x1][j];
  x=1;

  while((a[x1][j+NODE] != 1) && (j<NODE))   j++;
 /* printf("j=%d Node No. of each ss\n", j); */
  if(j!=NODE){
   while(x<=j){
    if(com[xy][cim] > a[x1][x]){
       cp=cim;
       while((com[xy][cim]>a[x1][x]) && (cim>=0)){
    temp=com[xy][cim];
    com[xy][cim]=a[x1][x];
    com[xy][cim+1]=temp;
         cim--;
         }
      cim=cp+1;
  /* printf("it is great than cim=%d x=%d\n", cim, x);  */
      }
      else{
      com[xy][++cim]=a[x1][x];
      }
     x++;
   } /* while loop */
  } /* if loop */
x1=flag[i+kin];
}   /* big for loop */
/*
if(nxn==1){
  for(x=0; x<NODE; x++){
 printf("com[0][x]=%d com[1][x]=%d\n", com[0][x], com[1][x]); }
}
```

```
554:    */
555:
556:    for(x=0; x<NODE; x++){
557:      if(com[0][x]!=com[1][x]) return(1);
558:     }
559:     return(0);
560:    }   /*compare function called from kill */
561:
562:
563:    /* to control the size of population, reproduce from the best */
564:    control()
565:    {
566:      int fa, sin;
567:
568:         fa=sin=0;
569:      printf("go to control loop $$$$$$$$$$$$$$, i=%d variety=%d\n", i, variety);
570:
571:          while(i<DATAF){
572:              sin=flag[fa];
573:               fa++;
574:            for(j=0; j<2*NODE; j++)
575:              {  a[i][j]=a1[sin][j];   }
576:             i++;
577:       }
578:       return;
579:     }   /* control function called from reproduce  */
580:
581:    /* this function is for redecompose the best gene, set to 10 */
582:      redo()
583:      {
584:          int re_a[50][50];
585:          int re_a1[50][30];
586:          int nx;
587:      /*  int re_NODE, re_temp, act_j1, re_conl, i1, nx, j1;  */
588:     /*
589:       printf("This is the last turn in redo\n");
590:       re_conl=0;
591:       i1=flag[0];
592:       re_NODE=15;
593:        for(j=0; j<2*re_NODE; j++){
594:         re_a[0][j]=0;
595:         }
596:         j1=j=0;
597:        re_a[0][j]=a[i1][j];
598:       for(j=1; j<re_NODE; j++){
599:       if(re_a[0][j1]>a[i1][j]){
600:          act_j1=j1;
601:        while((re_a[0][j1] > a[i1][j]) && (j1>=0)){
602:           re_temp=re_a[0][j1];
603:           re_a[0][j1]=a[i1][j];
604:           re_a[0][j1+1]=re_temp;
605:            j1--;
606:          }
607:         j1=act_j1+1;
608:         }
609:       else   re_a[0][++j1]=a[i1][j];
610:     }
611:    */
612:      for(i=0; i<variety; i++){
613:        for(j=0; j<30; j++)
614:          re_a1[i][j]=0;
615:       }
616:     for(i=0; i<variety; i++){
617:       for(j=0; j<15; j++)
618:          a[i][j]=j;
619:        }
620:
621:     for(i=0; i<variety; i++){
622:       j=0;
623:          while (j<=15){
624:             j = j+(int) 12*drand();
625:             re_a1[i][15-1+j] = 1;           /* if exceed the size? */
626:           }
627:         } /* generate the ss length for 15 node  */
628:
629:      a[0][15]=19;
630:      a[0][16]=18;
631:      a[0][17]=22;
632:      a[0][18]=21;
```

```
633:      a[0][19]=16;
634:      a[0][20]=15;
635:      a[0][21]=20;
636:      a[0][22]=17;
637:      a[0][23]=23;
638:      a[0][24]=24;
639:
640:      for(i=0; i<variety; i++){
641:        for(j=15; j<25; j++)
642:          a[i][j]=a[0][j];
643:       }
644:       for(i=0; i<variety; i++){
645:         for(j=15; j<30; j++)
646:            a[i][j+10]=re_a1[i][j];
647:       }
648:      a[0][40]=1;
649:      a[0][41]=1;
650:      a[0][42]=0;
651:      a[0][43]=0;
652:      a[0][44]=0;
653:      a[0][45]=0;
654:      a[0][46]=0;
655:      a[0][47]=1;
656:      a[0][48]=1;
657:      a[0][49]=1;
658:
659:    for(i=0; i<variety; i++){
660:       for(j=40; j<50; j++)
661:          a[i][j]=a[0][j];
662:      }
663:    for(i=0; i<variety; i++){
664:        for(j=0; j<15; j++)
665:          re_a1[i][j]=j;
666:      }
667:
668:      for(i=0; i<variety; i++){
669:        for(j=0; j<50; j++)
670:         re_a[i][j]=a[i][j];
671:      }
672:
673:     for(i=0; i<variety; i++){
674:        for(j=0; j<50; j++)
675:           a[i][j]=0;
676:       }
677:
678:      for(i=0; i<variety; i++){
679:         for(j=0; j<30; j++)
680:           a[i][j]=re_a1[i][j];
681:       }
682:
683:
684:     for(nx=0; nx<20; nx++){
685:      NODE=15;
686:       cross();
687:       for(i=0; i<variety; i++){
688:          for(j=0; j<30; j++)
689:          re_a1[i][j]=a1[i][j];
690:         }
691:       for(i=0; i<variety; i++){
692:         for(j=0; j<50; j++)
693:           a[i][j]=0;
694:        }
695:      for(i=0; i<variety; i++){
696:        for(j=0; j<15; j++)
697:          a[i][j]=re_a1[i][j];
698:        }
699:
700:       for(i=0; i<variety; i++){
701:          for(j=15; j<30; j++)
702:            a[i][j+10]=re_a1[i][j];
703:         }
704:      for(i=0; i<variety; i++){
705:        for(j=15; j<25; j++)
706:          a[i][j]=re_a[i][j];
707:        }
708:       for(i=0; i<variety; i++){
709:         for(j=40; j<50; j++)
710:          a[i][j]=re_a[i][j];
711:        }
```

```
      NODE=25;
    for(i=0; i<variety; i++){
        for(j=0; j<2; j++)
         Tran[i][j]=0;
    }
     transition();
    printf("this is after transition\n");
    sort();
     kill();
    for(i=0; i<variety; i++){
       for(j=0; j<50; j++)
        a[i][j]=0;
    }
    for(i=0; i<variety; i++){
      for(j=0; j<30; j++)
       a[i][j]=re_a1[i][j];
      }
    NODE=15;
   reproduce();
  printf("nx=%d\n", nx);
  }

   return;
  }  /* function of redo */


  /* reproduce the best data according to the best transition */
  reproduce()
 {
    int ind1, ind, y1, con, copy;
    for(i=0; i<variety /*50*/; i++){
       for(j=0; j<2*NODE; j++)
        a1[i][j] = a[i][j];
      }

      for(i=0; i<variety; i++){
        for(j=0; j<2*NODE; j++)
           a[i][j]=-1;
    }

    i=con=y1=ind=ind1=0;
     copy=7;

    while(i<DATAF)
     {/* reproduce the first 15 data from 16--1 copies by decr */
      ind1=flag[ind];
      con=con+copy;
      for(i=y1; i<con; i++){
        for(j=0; j<2*NODE; j++)
      a[i][j]=a1[ind1][j];
        }       /* for */
       y1=i;
      ind++;
        if((copy !=1) && (copy>0))  copy--;
       if((impo<49) && (ind==impo+1)){
              control();
         }
      }

     for(i=0; i<variety; i++){
        for(j=0; j<2*NODE; j++)
            a1[i][j]=-1;
      }

   return;
  }    /*reproduce  function */

 /* this function serve the purpose of inhibiting the node which geat tan 14 */
 re_cal()
  {
     int xx, re_m, re_id;

       re_id=15;
       re_m=0;
    for(xx=0; xx<5; xx++){
      while((re_id != B[re_m]) && (B[re_m] != -1))  re_m++;
       if(re_id==B[re_m]){
           B[re_m]=B[total-1];
           B[total-1]=-1;
```

```
            total--;
            cn--;
          }
         re_id++;
        }
     return;
    }

        /*********************************************************************/
   /* 6) cross over to generate new data */
   cross()
   {
     struct repeat{
                   int no;
                   int position[30];
                  }o[25];
     int x1, y1, np, nc, nd, z1, f, temp, m,nz;
     int l, k, n2, r,t, q, yy, re, g;
     int napp[25];
     yy=0;
     q=variety-1;

   while(q>=0){
     re=1;

     for(y1=0; y1<NODE; y1++){
        o[y1].no = 0;
       for(x1=0; x1<30; x1++)
        o[y1].position[x1] = -1;
        }
    while((re==1) && (q>1)){
     z=(q+1)*drand();
     z1=q*drand();      /* DATAF substitute by 70 */
    if((z==z1) || ((z1+1)==q) || (z==q) || ((z+1)==q)) re=1;
    else re=0;
   }
    if(q==1){
     z=1;
     z1=0;
    }

   /*if(nxn==0){
      printf("next gene cross z=%d z1=%d q=%d\n", z, z1, q);
    }*/
     j=f=0;
     while(j<NODE)
     {
       if(f != -1)
       {
          while((a[z1][j+NODE] ==0) && (a[z][j+NODE]==0))
             {j++;}
      temp=a[z][++j];
           a[z][j]=a[z1][j];
           a[z1][j]=temp;
       /*   printf("first cross j=%d\n", j); */
        }
      if((a[z][j+NODE]==0) && (a[z1][j+NODE]==0))
       {
         temp=a[z][++j];
         a[z][j]=a[z1][j];
         a[z1][j]=temp;
         /* printf("continue j=%d\n", j); */
         f=-1;
        }

       else{
        temp=a[z][j+NODE];
        a[z][j+NODE]=a[z1][j+NODE];
        a[z1][j+NODE]=temp;
        f=0;
        j++;
        }
     }
   /* check if new generated data repeated or not */

     for(n2=0; n2<NODE; n2++){
       napp[n2]=-1;
     }
     nz=z;
```

```
870:      for(y1=0; y1<=1; y1++)
871:      {
872:       for(x1=0; x1<NODE; x1++){
873:         napp[x1]=-1;
874:         o[x1].no=0;
875:               for(np=0;np<30; np++)  o[x1].position[np]=-1;
876:       }  /* init structure and other para */
877:
878:      j=m=0;
879:      l=-1;
880:        while(j < NODE)
881:        {
882:          n2=a[nz][j];
883:          o[n2].no++;
884:          j++;
885:        }   /* record the number of nodes appear */
886:
887:        for(j=0; j<NODE; j++){
888:          n2=a[nz][j];
889:          if(o[n2].no==1)  o[n2].position[0]=j;
890:        }   /* record the position of node which appear once */
891:
892:         for(n2=0; n2<NODE; n2++){
893:       if(o[n2].no>1){
894:              k=j=0;
895:              for(x1=0; x1<o[n2].no; x1++){
896:                 while(n2 != a[nz][j])
897:                  { j++; }
898:                 o[n2].position[k]=j;
899:                 k++;
900:                 j++;
901:               }
902:       }
903:          }  /* record the position of the node which appear more than once */
904:
905:         n2=0;
906:         while(n2<NODE)
907:         {
908:           if(o[n2].no==0)
909:           {
910:        l++;
911:        napp[l]=n2;
912:           }
913:           n2++;
914:         }  /* record the node which not appear in the gene */
915:
916:       /*  for(n2=0; n2<l; n2++)  printf("n2=%d napp[n2]=%d\n", n2, napp[n2]); */
917:         for(n2=0; n2<NODE; n2++)
918:         {
919:           k=1;
920:           while(o[n2].no >1)
921:           {
922:           m=o[n2].position[k];
923:           r=(l+1)*drand();
924:           a[nz][m]=napp[r];
925:              /* printf("n2=%d r=%d m=%d napp[r]=%d l=%d\n", n2,r,m,napp[r], l);*/
926:             if(r != l){
927:               napp[r]=napp[l];
928:               napp[l]=-1;
929:         }
930:           l--;
931:           o[n2].no--;
932:           k++;
933:             } /* while */
934:         }  /* for loop, replace the node with non-appear node */
935:
936:         nz=z1;
937:       /*printf("z=%d z1=%d nz=%d\n", z, z1, nz);*/
938:       }  /* for (y1)   */
939:
940:     /* check the new generated data illegality */
941:      nz=z;
942:       for(n2=0; n2<80; n2++) B[n2]=-1;
943:
944:      for(x1=0; x1<=1; x1++){
945:       j=k=t=cn=l=nd=0;
946:         while(j<(NODE-1))
947:         {
948:           l=0;
```

```
949:         g=a[nz][j];
950:         n2=n[g].no;
951:         cn=cn+n2;
952:         t=j;
953:       /*  printf("A NEW g=%d n2=%d j=%d\n", g, n2, j); */
954:           while(a[nz][j+NODE] !=1 && j < NODE-1)
955:           {
956:              if(l != -1)
957:                  {
958:                k=nd=0;
959:           while(nd< cn)
960:             {
961:              B[nd]=n[g].nb[k].id;
962:              k++;
963:              nd++;    /* nd is the total no of neig */
964:              }
965:                   }
966:              if(nxn==41)
967:              {
968:                total=nd;
969:                re_cal();
970:                nd=total;
971:               }
972:
973:             if((a[nz][j+NODE]) != 1){
974:               g=a[nz][++j];
975:               m=0;
976:               /*printf("next g=%d j=%d t=%d a[t]=%d\n", g, j, t, a[nz][t]);
977:                 for(y1=0; y1<nd; y1++) printf("y1=%d B[y1]=%d\n", y1, B[y1]);*/
978:                     while((g != B[m]) && (B[m] != -1))
979:              {   m++; }
980:             if(g==B[m])
981:             {
982:                    if(m != (nd-1)){
983:                   B[m]=B[nd-1]; }
984:              B[nd-1]=-1;
985:                   nd=nd-1;
986:                   cn=cn-1;
987:              temp = a[nz][j];
988:              a[nz][j]=a[nz][++t];
989:              a[nz][t]=temp;
990:              n2=n[temp].no;
991:              cn = cn+n2;
992:              j=t;
993:                     l=0;
994:                /*  printf("find g=%d t=%d a[nz][t]=%d cn=%d nd=%d\n", g, t, a[nz][t], cn, nd); */
995:              }
996:              else  l=-1;
997:                 }    /* if this ss is finish */
998:               }   /* while  a new germ begins */
999:           /*  printf("t=%d j=%d\n", t, j); */
000:        a[nz][t+NODE]=1;
001:             j=t+1;
002:             for(nd=0; nd<80; nd++)
003:                  B[nd]=-1;
004:               k=l=cn=nd=0;
005:           } /* while j<NODE */
006:
007:      for(j=0; j<2*NODE; j++){
008:        a1[yy][j]=a[nz][j];
009:       }
010:       yy++;
011:     nz=z1;
012:     }  /* for a new germ */
013:       for(x1=0; x1<2; x1++){
014:        for(y1=nz; y1<q; y1++){
015:          for(j=0; j<2*NODE; j++)  a[y1][j]=a[y1+1][j];
016:         }
017:         q--;
018:         nz=z;
019:       }
020:     /* printf("before return yy=%d q=%d\n", yy, q);   */
021:     }  /* while */
022:
023:       for(i=0; i<variety; i++){
024:          for(j=0; j<2*NODE; j++)
025:            a[i][j]=-1;
026:        }
027:     return;
```

```
028:    }   /*cross over function */
```

Appendix B

Reference

[1] L. Kleinrock, *"Queueing Systems"*, vol I, Wiley-Interscience, New York, 1975.

[2] T.G.Robertazzi, I.Y.Wang, *"Recursive Compuatation of Steady State Probabilities of non-Product form queueing networks associated with computer network protocols."*, 30th Midest Symposium on Circuits and Systems, Syracuse, N.Y., Aug. 1987.

[3] W.J. Gordon, G.J. Newell, *"Closed Queueing Systems with Exponential Servers"*, Operations Research, 15 March, 1967.

[4] T.G. Robertazzi, *"Automatic Decomposition of Certain Non-Product Form Protocol Models"*, CEAS Technical Report 504, March 29, 1988. Accepted by IEEE Transactions on Communications.

[5] J.P.Bnzen, *"Computational Algorithms for Closed Queueing Networks with Exponential Servers"*, Comm. ACM, vol, 16, No. 9, Sept. 1973.

[6] John Holland, *"Genetic Algorithm"*, 1989

1. J.R.Jackson, *"Networks of Waiting Lines"*, Operations Research, 5, 1957.

[7] M.Schwartz, *"Telecommunication Networks Protocols, Modeling and Analysis"*, Addison-Wesley 1987.

[8] J.E.Baker, *"Adaptive Selection Methods for Genetic Algorithms"*, Proceedings of An International Conference on Genetic Algorithms and their Applications, Pittsburgh PA., July 1985.

[9] D.E. Goldberg, *"Genetic Algorithms in Search, Optimization, and Machine Learning"*, Addison Wesley, 19889.

[10] J.D. Schaffer and A. Morishima, *"An Adaptive Crossover Distribution Mechanism for Genetic Algorithm"*, Proceedings of second International Conference on Genetic Algorithms, Cambridge MA., July 1987.

[11] J. Grefensstette, R.Gopal, B.Rosmaita and D.V. Gucht, *"Genetic Algoritjms for the Travelling Salesman Problem"*, Proceedings of An International Conference on Genetic Algorithms and Their Applications, Pittsburgh PA., July 1985.

[12] D.E. Goldberg and R. Lingle, Jr., *"Alleles, Loci, and the Travelling Salesman Problem"*, Proceedings of An International Conference on Genetic Algorithms and Their Applications, Pittsburgh PA., July 1985.

[13] C.K. Hong and I.Y. Wang, *"A Decomposition Algorithm for the State Transition Lattices of Certain Non-Product Queueing Protocols"*, Proc. of the 27th Allerton Conference, Urbana-Champaign IL., Sept 1989.