Fall 1-31-1991

Decomposing non-product form queueing lattices through autorouting with A* algorithm

Chun-Chang Yu
New Jersey Institute of Technology

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

Decomposing Non-Product Form Queueing Lattices Through Autorouting With A * Algorithm

BY *CHUN-CHANG YU*

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

1991

VITA

NAME : CHUN-CHANG YU

Address : 125 Lincoln Ave. Apt. B5 Newark, NJ 07104

Eductation :

1. New Jersey Institute of Technology
   Electrical Engineering Department, M.S.E.E.
   Spring,1989 - January,1991
2. Tamkung University
   Electrical Engineering Department, B.S.E.E.
   Fall,1982 - Fall,1987

Position Held :

1. Digital Equipment Taiwan LTD
   Testing Engineer
   December 1987 - July 1988
2. Fortune Taiwan CO.
   Circuit Design Engineer
   July 1986 - November 1987

ABSTRACT

TITLE OF THESIS :

A Simulation to Decompose Non-Product Form Queueing Lattices Through Autorouting with A* Algorithm

Name:

Chun-Chang Yu

Master of Science in Electrical Engineering

Thesis Directed by : Dr. Irving Wang

Direct solution techniques are expensive for the state transition lattice of a class of non-product form queueing modes. In this thesis, autorouting with A* algorithm is adapted to decompose it as solvable subsets which can be solved sequentially and independently. Autorouting with A* algorithm is heuristics in problem solving. It also is one of a class of global optimization problems that are difficult to solve. Some queueing modes of type A will be decomposed in this thesis through aAutorouting with A* algorithm.

ACKNOWLEDGEMENTS

I take this opportunity to express my deep gratitude to Dr. Irving Y. Wang, Assistant Professor, Electrical Engineering Department of N.J.I.T. for his advice and help throughout the course of this work.

I am also very thankful to Dr. Nirwan Ansari and Dr. Edwin Hou for devoting their precious time to reviewing my work and attendance.

Finally, I wish to give my loving appreciation for the encouragement and support given to me by my parents.

Approval Sheet

Title of Thesis: **Decomposing Non-Product Form Queueing Lattices Through Autorouting With A * Algorithm**

Name of Candidate: Chun-Chang Yu
Master of Science in Electrical Engineering, 1990

Thesis & Abstract Approved
by the Examining Committee:

Dr. Irving Wang, Advisor — Date
Assistant Professor
Department of Electrical and Computer Engineering

Dr. Nirwan Ansari — Date
Assistant Professor
Department of Electrical and Computer Engineering

Dr. Edwin Hou — Date
Assistant Professor
Department of Electrical and Computer Engineering

New Jersey Institute of Technology, Newark, New Jersey.

Contents

Appendix:

List of Figures

Chapter 1
Introduction

In studying the behavior of computer system, queueing network modes are a useful tool for evaluating performance and for studying the interaction between 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 modes. Product form solutions are of great interest because the direct numerical solution of balance equations is computationally expensive. Typically the existence of product form solutions has been characterized in terms of certain classes of queueing networks. In [2,3,4]it was characterized in terms of the algebraic topology of the state transition lattice. 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.

Non-product form queueing networks are far less likely to have a closed form solution for the equilibrium probabilities. Direction solution techniques are prohibitively expensive. In this paper autorouting with A* algorithm will be submitted to decompose the non-product queueing lattice.

In chapter 2, the product form and the non-product will be reviewed, while at the same time sequential decomposition will be discussed. In chapter 3, the autorouting with A* algorithm is described. In chapter 4, the algorithm discussed in chapter 3 is implemented to decompose the non-product form queueing lattice for A type. Chapter 5 shows the results of simulation.

Chapter 2
Product and Non-Product Form

2.1 General Review

In [4] 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 time[5]. These probabilities were of a characteristic "product form":

$$P(n_1 n_2,\ldots,n_m) = P_1(n_1)P_2(n_2)\ldots P_m(n_m) \qquad (1.1)$$

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

$$P(n_1,n_2,\ldots,n_m) = \frac{1}{G(N)}\prod_{i=1}^{m} f_i(n_i) \qquad (1.2)$$

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).

There is 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[6] rather than current. Transition rates 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 directions are perdetermined from the transition directions. Finally, the transition rates are labeled in a patterned manner from the queueing scheme. 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 as shown in fig. 2.1.

Non-product is shown in fig 2.2. Direct solution techniques are prohibitively expensive. Techniques analogous to that of the z-transform can sometimes be used to determine distributions of interest[4,7]. In[7] the sequential decomposition technique was described.

Figure 2.1a

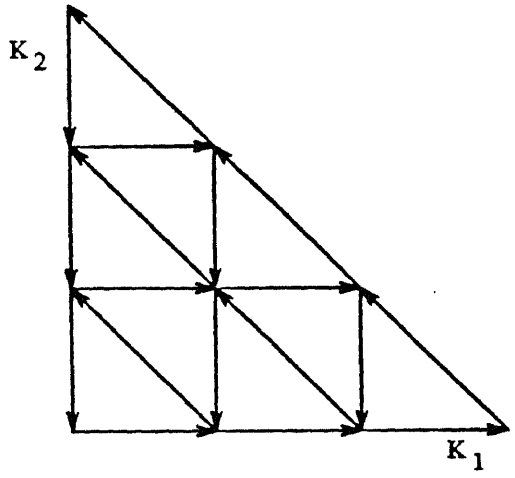


Figure 2.1b

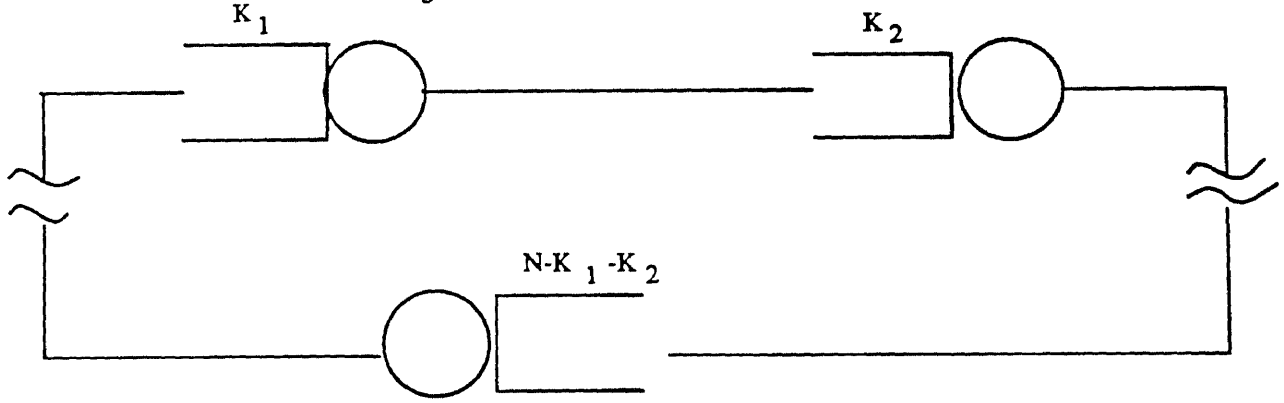


Figure 2.1c

Figure 2.1 Product Form

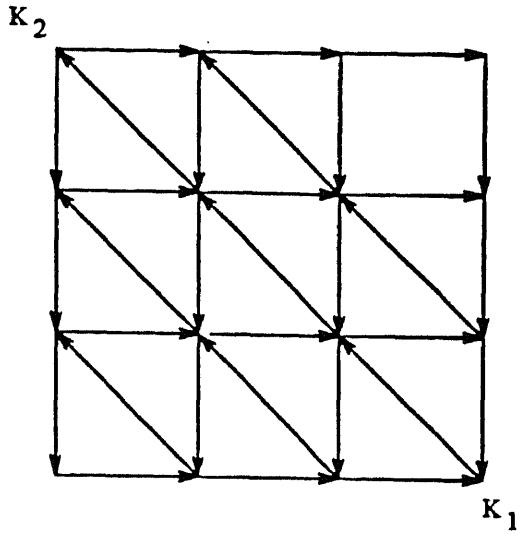


Figure 2.2 Non-Product Form

2.2 Sequential Decomposition

The exact computation of the state probabilities of a class of non-product form queueing networks is discussed in this section. 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[4] 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 subset 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 regard 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 figure 2.3. States S1 and S2 form a solvable subset. The equilibrium probabilities can be determined without regard to the values associated with the other states. This is done through the following global balance equations:

$$(\alpha+\beta)\ P_0 = \gamma P_1 + \beta P_2$$

$$\alpha P_2 = \gamma P_1$$

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 figure 2.4. 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 form which a transition(s) entering the cluster originates.

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,....,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 thanN^3.

Two types of the structure which allows the state transition lattice to be decomposed into solvable subsets were obtained in [7]. The first type of structure is illustrated in Figure 2.4 and Figure 2.6. 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,.... , subsets. This type of structure is type A structure.

The second type of structure is illustrated in Figure 2.5. Here the first subset consists of a single state. The remaining subsets each consist of a state or a group of states. They are arranged in a tree 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. 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 is referred to as type B structure.

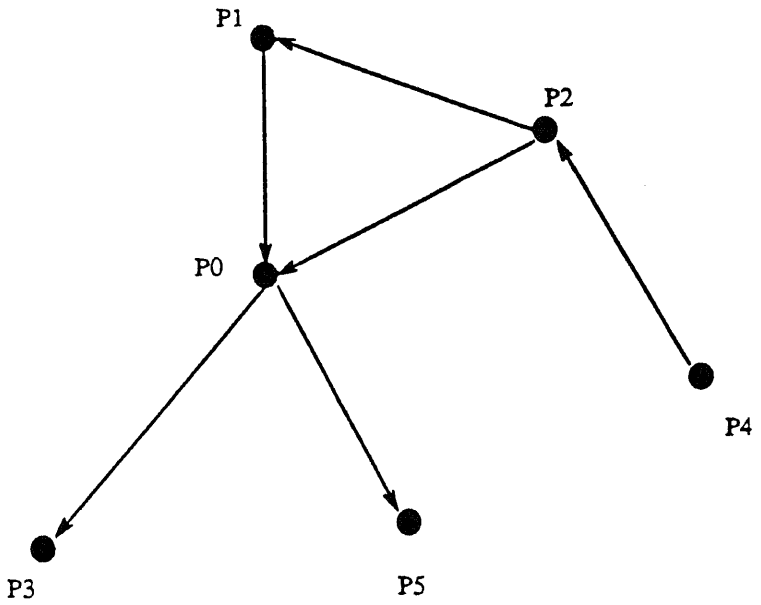


Figure 2.3 A Solvable Subset

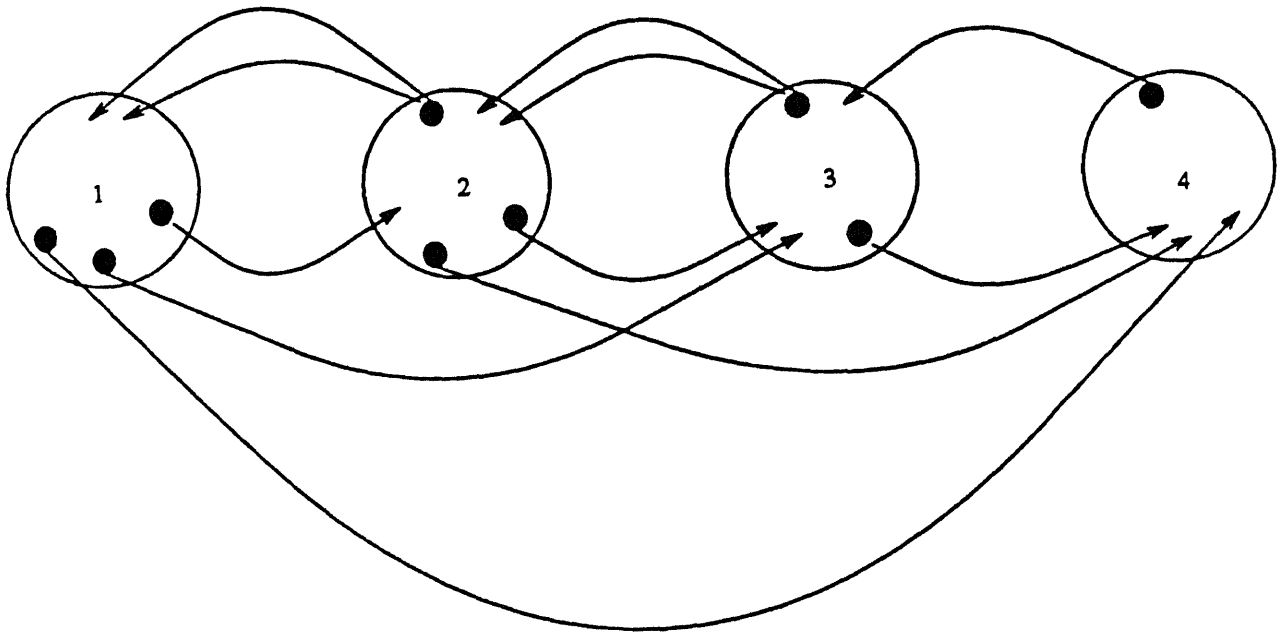


Figure 2.4 The Basic Structure of A Type A

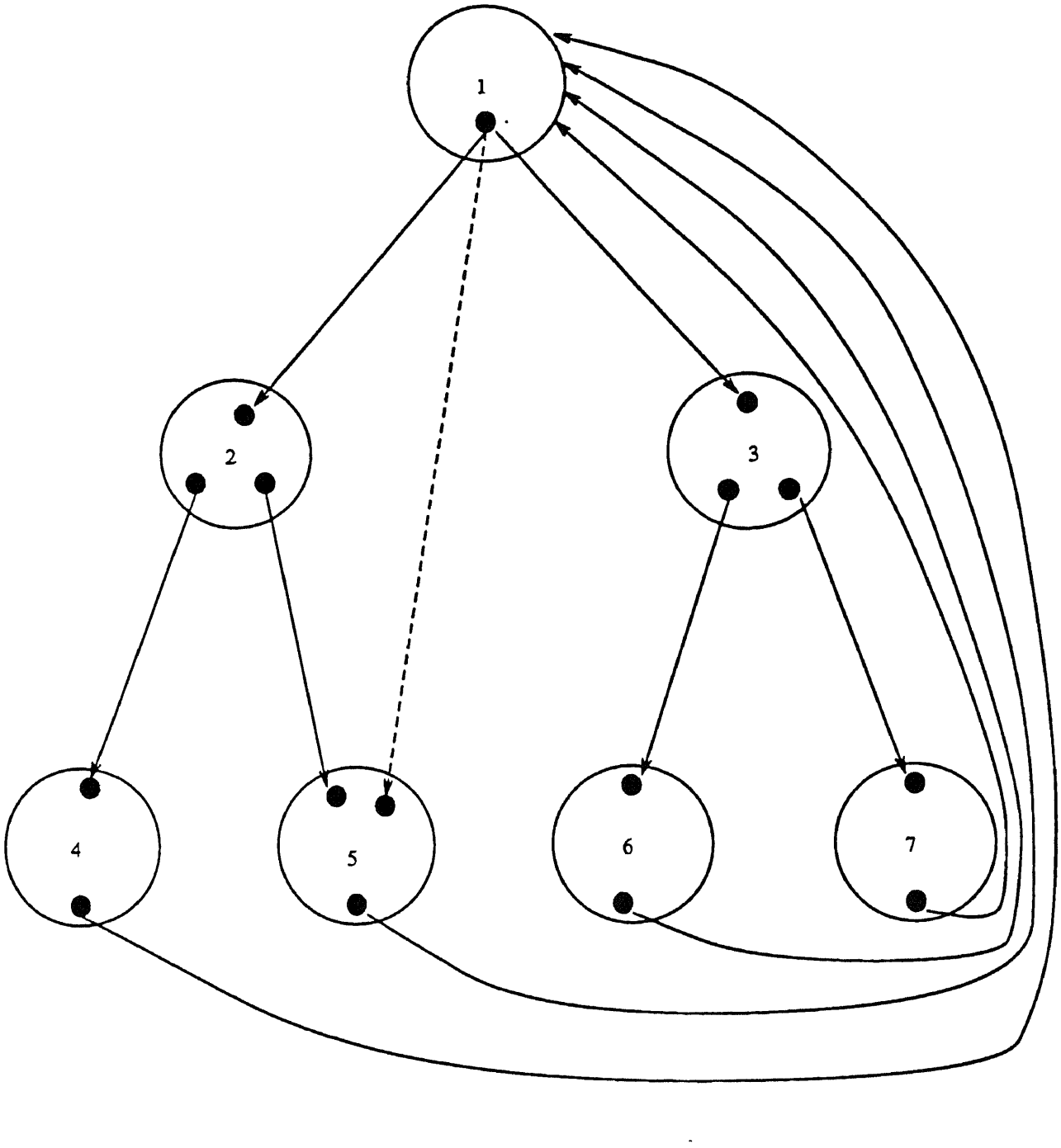


Figure 2.5 The Basic Structure of Type B

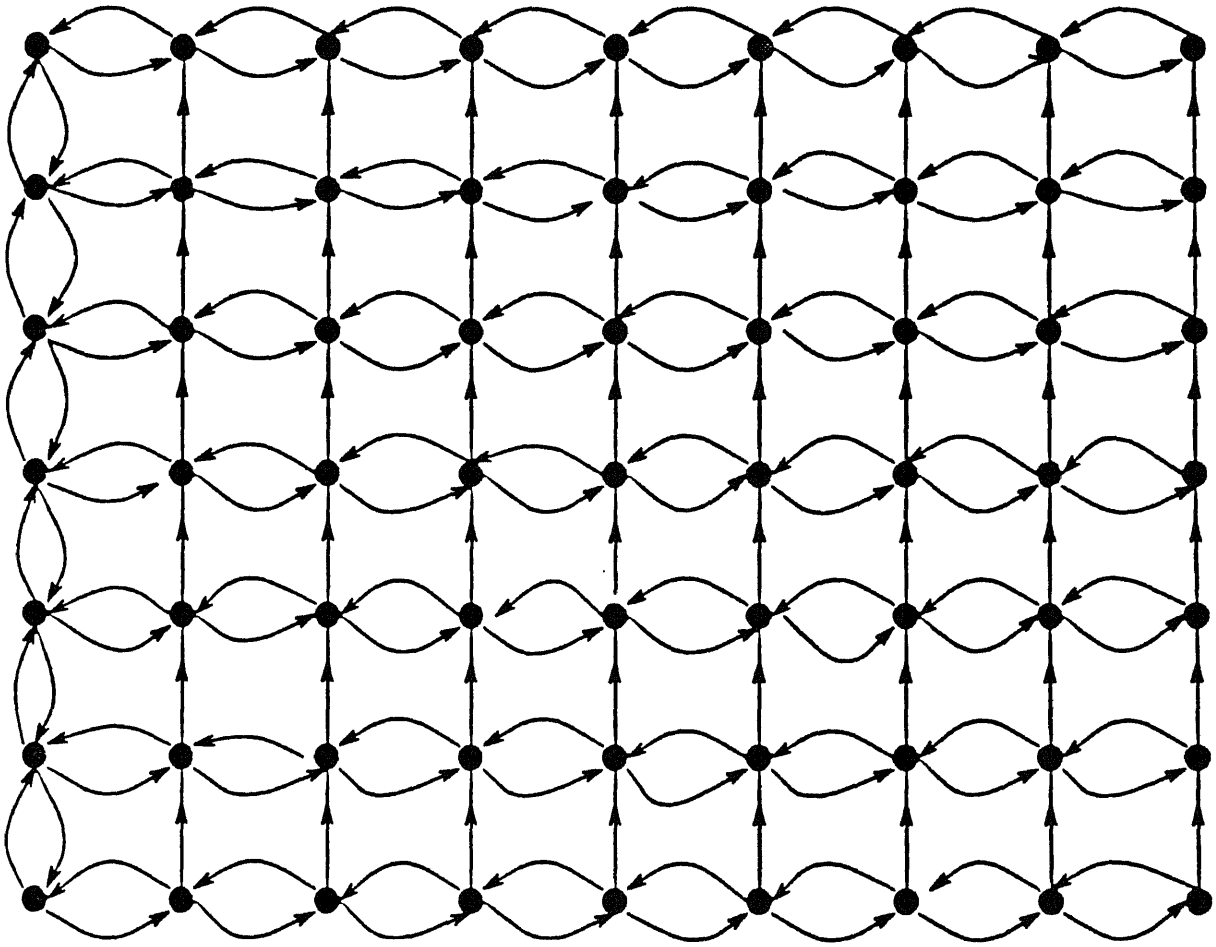


Figure 2.6 A Typical Example of Type A Lattices

Chapter 3
Autorouting with the A* Algorithm

Autorouting is one of a class of global optimization problems that are difficult to solve. A good decomposition for the non-product form queueing networks, for example, minimizes things like:

- the number of groups of states
- the time to search the group

At the same time, the decomposition maximizes things like reliability and ease of debugging. The overall value of a decomposition for a non-product form queueing network design is a function of these often conflicting variables. It is usually acceptable to find a solution for all but the most trivial problems. In this chapter, we would describe the autorouting with the A* algorithm about how it works in the two phases and analysis of its complexity.

3.1 Autorouting Operation

Autorouting search algorithms typically operate in two phase[9], treating the state-transition lattice as a matrix of cells. The first phase starts at the source cell and searches for the target cell, usually by going in several directions at the same time .

The algorithms build in an auxiliary data structure to keep track of how each cell was reached. The first phase ends when the target cell has been found, and the second phase begins. If the first phase exhausts all possibilities without reaching the target cell, then no route exists between them, and there is no reason to do the second phase.

The second phase uses the auxiliary data structure to trace the route from the target cell back to the source cell.

The second phase is identical for the breadth-first [10][11] and A* search[12] algorithms. The first phase is different, and it is this phase that gives these algorithms different behaviors.

3.1.1 Data Structure for the First Phase

The main data structures used in the first phase are the OPEN QUEUE and the CLOSED SET, which hold cell coordinates. Because a cell's coordinates uniquely identies it, we'll say that the OPEN QUEUE and CLOSED SET contain cells. Cell corrdinate will be represented as r5c5 for the cell at row 5, column 5.

To remind ourselves that OPEN is a queue and CLOSED is a set, when we talk about adding cell to them, we will put the cells " on " the queue and " in " the set. Initially, the OPEN QUEUE contains the source cell and the CLOSED SET is empty.

The first phase is a loop which removes an element from the head of the OPEN QUEUE, puts it in the CLOSED SET (which indicates that it has been searched) and checks to see if it is the target cell. If so, the first phase is done, otherwise, the neighbors of the cell (those adjacent to it) are placed on the OPEN QUEUE, and the loop continues. As we'll see later, the essential difference in the breadth-first and A* search algorithms is placed on the OPEN QUEUE.

3.2 Breadth-First Search

Breadth-first search looks for the target among all elements of the matrix from the source cell.

Here is the procedure with proper terminating condition

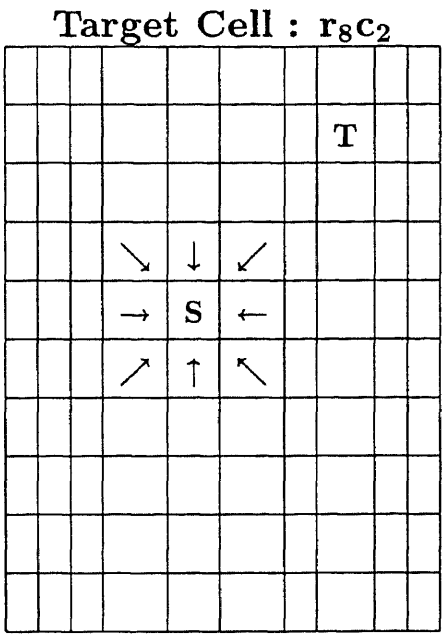
To conduct a breadth-first searth:

1. From a one cell set consisting of the source.
2. Until the QUEUE is empty or the goal has been reached, determine if the cell(s) in the set is(are) the target.
   (a) If the set is the target, do nothing.
   (b) If the set is not the target, put its neighbors to the SET and add the neighbors of their neighbors to the QUEUE. Go back to 2a.
3. If target has been found, announce success; otherwise announce failure.

Breadth-first search works by processing a first in first out (FIFO) queue of open cells, that is, cells that have been reached, but not yet searched. Initially, the OPEN QUEUE contains only the source cell. A cell is removed from the head of the OPEN QUEUE, placed in the SET of closed cells (cells that have been searched) and checked to see if it is the target cell. If not, its neighbors are placed at the tail of the OPEN QUEUE. Neighboring cells that have already been reached are ignored (If a cell's corrdinates is on the OPEN QUEUE or in the CLOSED SET, then it has been reached, otherwise it has not). This continues until one of two things happens:

- The goal cell has been found.
- The OPEN QUEUE is empty in which case the goal can not be reached from the source cell.

A version of breadth-first search known as Lee's algorithm[11] has served as the basis for some autorouters since the early 1960s.



CLOSED OPEN

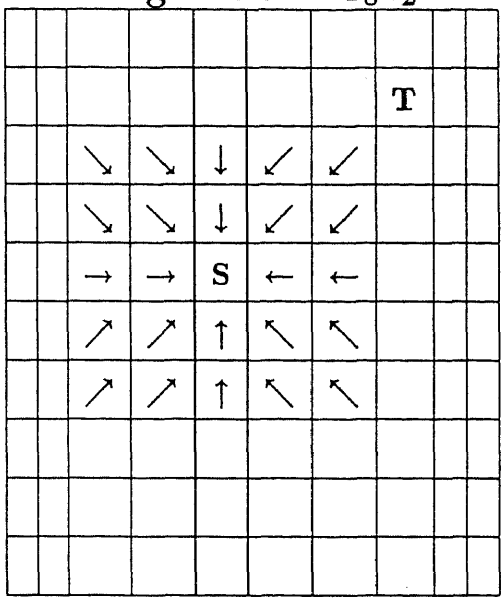
r_5c_5 r_5c_4 r_6c_5 r_5c_6

r_4c_5 r_6c_4 r_6c_6

r_4c_6 r_4c_4

Figure 3.1a

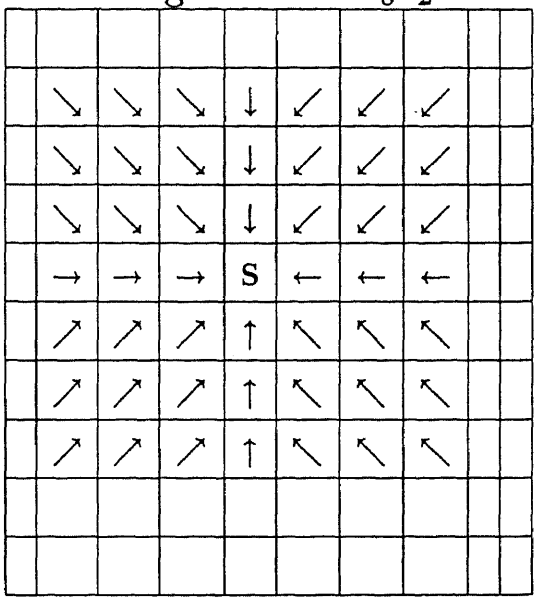
Target Cell : r_8c_2



CLOSED		OPEN		
r_5c_5	r_5c_4	r_5c_3	r_6c_3	r_7c_3
r_6c_5	r_5c_6	r_7c_4	r_7c_5	r_7c_6
r_4c_5	r_6c_4	r_7c_7	r_6c_7	r_5c_7
r_6c_6	r_4c_6	r_4c_7	r_3c_7	r_3c_6
r_4c_4		r_3c_5	r_3c_3	r_3c_4
		r_4c_3		

Figure 3.1b

Target Cell : r_8c_2



CLOSED OPEN

r_5c_5 r_5c_4 r_6c_5 r_5c_2 r_6c_2 r_7c_2 r_8c_2

r_5c_6 r_4c_5 r_6c_4 r_8c_3 r_8c_4 r_8c_5 r_8c_6

r_6c_6 r_4c_6 r_4c_4 r_8c_7 r_8c_8 r_7c_8 r_6c_8

r_5c_3 r_6c_3 r_7c_3 r_5c_8 r_4c_8 r_3c_8 r_2c_8

r_7c_4 r_7c_5 r_7c_6 r_2c_7 r_2c_6 r_2c_5 r_2c_4

r_7c_7 r_6c_7 r_5c_7 r_2c_3 r_2c_2 r_3c_2 r_4c_2

r_4c_7 r_3c_7 r_3c_6

r_3c_5 r_3c_4 r_3c_3

r_4c_3

Figure 3.1c

Lee's algorithm searching for a path

In Figure 3.1a, the source cell (r5c5) has been searched, and its eight neighbors have been placed on the OPEN QUEUE. The arrows indicate the direction from which each cell was reached, and correspond to first phase data structure. After the first eight cells on the OPEN QUEUE have been reached and moved to the CLOSED SET, the algorithm searches the configuration in Figure 3.1b, where there are sixteen cells on the OPEN QUEUE. Once these sixteen cells have been searched, the configuration in Figure 3.1c is reached. Now the goal cell (r8c2) is fourth from the end on the OPEN QUEUE, and a solution is imminent.

3.3 A* Algorithm Search

The A* procedure is branch-and-bound search, with an estimate of remaining distance, combined with the dynamic-programming principle [10]. It is a priority queue, which means cells are inserted according to the estimated distance to the target [10], not just at the end.

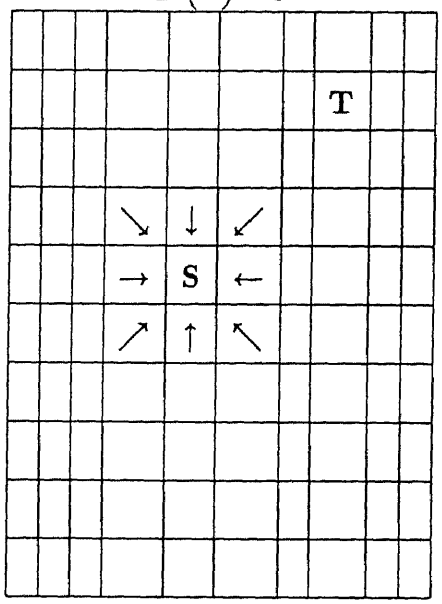
Cells that are on the shortest estimated path from source to target go to the head of the queue. The A* algorithm removes the cell from the head of the OPEN QUEUE and checks to see if it's the target. If not, the neighboring cells are put on the OPEN QUEUE at the proper position. The algorithm checks neighboring cells that have already been searched to see if the new path between them and the source is shorter than the previous one. If it is, they are repositioned on the OPEN QUEUE according to the new estimated path length from source to goal. As in Breadth-First search, this continues until the target cell has been found or until the OPEN QUEUR is empty.

A* depends on being able to estimate the distance between a cell and the target cell. In the case of autorouting, a simple measure of this distance is available, and this helps A* to concentrate the search on the direction most likely to succeed. The more accurate the estimate, the faster the search.

Figure 3.2 shows the behavior of the A* search algorithm. A* does not specify whether new cells go in front of or behind cells already on the OPEN QUEUE that evaluate to identical estimated path lengths. We use the convention that they are placed in front . This minimizes the time to insert a cell on the OPEN QUEUE.

Target Cell : r_8c_2

$F(x)$=3

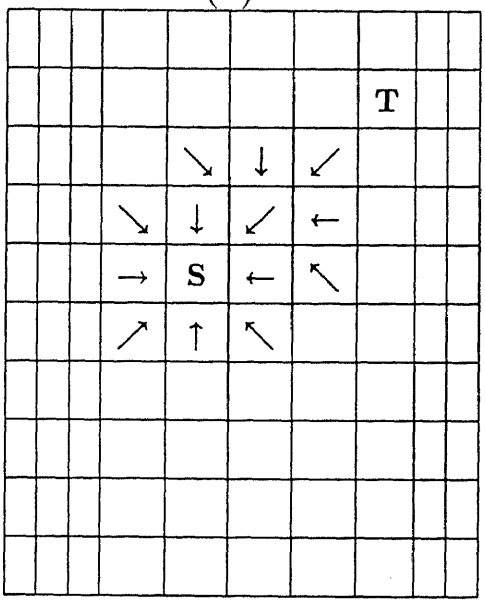


CLOSED	OPEN			
r_5c_5	$r_5c_4(3)$	$r_6c_5(3)$	$r_5c_6(4)$	$r_4c_5(4)$
	$r_6c_4(2)$	$r_6c_6(4)$	$r_4c_6(4)$	$r_4c_4(4)$

Figure3.2a

Target Cell : r_8c_2

$F(x)$=**2**

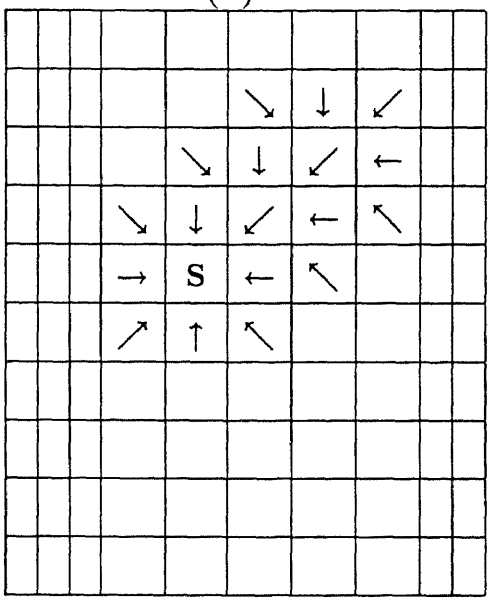


CLOSED		OPEN		
r_5c_5	r_6c_4	$r_5c_4(3)$	$r_6c_5(3)$	$r_5c_6(4)$
		$r_4c_5(4)$	$r_6c_6(4)$	$r_4c_6(4)$
		$r_4c_4(4)$	$r_6c_3(2)$	$r_7c_4(2)$
		$r_7c_3(1)$	$r_7c_5(3)$	$r_5c_3(3)$

Figure 3.2b

Target Cell : r_8c_2

$F(x)$=1



CLOSED		OPEN		
r_5c_5	r_6c_4	$r_5c_4(3)$	$r_6c_5(3)$	$r_5c_6(4)$
r_7c_3		$r_4c_5(4)$	$r_6c_6(4)$	$r_4c_6(4)$
		$r_4c_4(4)$	$r_6c_3(2)$	$r_7c_4(2)$
		$r_7c_5(3)$	$r_5c_3(3)$	$r_7c_2(1)$
		$r_8c_3(1)$	$r_8c_2(0)$	$r_8c_4(2)$
		$r_6c_2(2)$		

Figure 3.2c

Behavior of the A* search algorithm

In Figure 3.2a, the source cell (r5c5) has been searched, and its neighbors are on the OPEN QUEUE. Each cell on the OPEN QUEUE also includes the estimated length of the shortest path from S to T that goes through that cell (we set the estimated length of each grid to be one). After the first cell (r6c4) has been searched and moved to the CLOSED SET, the configuration in Figure 2.2b is reached, where there are 12 cells on the OPEN QUEUE. After searching the next cell (r7c3), the algorithm reaches the configuration in Figure 3.2c. Now the goal cell (r8c2) is at the head of the OPEN QUEUE, and solution will be found on the next iteration of the loop. Searching r8c2, A* recognizes it as the target and uses the first phase data structure to construct a trace back to the source cell.

3.4 Discussion

Lee's algorithm suffers from a behavior inherent in the breadth-first search technique , which limits its application to problems of relatively small size. As the distance between the source and target cells increases by a factor of N, the number of cells processed by Lee's algorithm –and therefore processing time – increases by the square of N.

A* algorithm search process, as seen in the Figure 3.2, goes more directly toward the goal cell. The target draws the search much as the earth's gravity pulls objects toward the center of mass. If we double the size of the problem, the search will process about twice as many cells, and if we triple its size, the search will run through three times as many. This linear characteristic makes A* more attractive for autorouting than the quadratic Lee's algorithm.

Chapter 4
Adapted Autorouting with A* Algorithm to Non-Product Form Queueing Mode

In chapter 3 the autorouting with A* algorithm operation is introduced. In this chapter it will be implemented to decompose the non-product form queueing lattices. The non-product form queueing lattices will be seen as a matrix. From a given cell, we will discuss how the target(group) will be found in this chapter . The pseudocode will also be supplied.

4.1 Type A

Type A structure is illustrated in Figure 2.4. Here each circular subset represents a state or a group of states. For the ith subset there must be only one state with unknown probability, external to the subset from which a transition(s) entering the subset originates. Our target is to find a small subgroup of cells that only one cell, not contained in the target group, can enter this subgroup from the source cell .

From Figure 2.4 we can find that, at most, only one node enters the subgroup from other subgroups . For convenience we call this node the bridge node. Now we give every node an estimate . The rule is that the estimates of the cells of the former subset should be less than that of the cells of the latter subset , i.e. the estimates of the nodes of the second subset should be greater than those of the nodes of the first subset. After each subset has been searched, i.e. after the bridge node for the subset has been found , the every bridge node will be seen as the source cell for the search of the other subgroups. Repeat this until we have done with each node.

If a source cell is a subset we have just finished with, set this subset's neighbors as the bridge cell and search again. If not, put the source cell in CLOSED SET, add its neighboring nodes that have path and compare their estimates to search for the cell with largest estimate as a bridge node until there is only one node i.e. the bridge cell , left on the OPEN QUEUE. Search the other subgroups again until no node can be found.

Pseudo code for searched subgroup from type A

(* S is the start node *)

(* OPEN is an ordered list of nodes , CLOSED is a set of nodes (order contain the lower F value) . In general, nodes that need to be searched are put on OPEN (at the proper position) . As they are searched, they are removed from OPEN ann put in CLOSED *)

(* $F(x)$ is the estimated of x *)

(* $G(x)$ is the number of nodes that enter the x, from its neighbors *)

1 OPEN $\leftarrow$ { S } (* a list of one element *)

CLOSED $\leftarrow$ { } (*the empty set *)

while OPEN $\neq$ { } dand not found do

x $\leftarrow$ the first node on OPEN (* node with largest F valus *)

OPEN $\leftarrow$ - {x} (*remove x from OPEN *)

CLOSED $\leftarrow$ CLOSED + {X}(* put x in CLOSED *)

5 IF $G(X) == 1$ (* x itself is a subset *)

Let N be the set of neighboring nodes of x

For each y on N do

IF($G(x) = \{ \}$) do nothing

IF ($G(X) == 1$) (* x iteslf is subset *)

S $\leftarrow$ y

$F(x) \leftarrow F(y)$

$G(x) \leftarrow G(y)$

Print CLOSED (* searched finished *)

go to 1

ELSE

IF y not on OPEN or CLOSED THEN

IF $F(y)$ *l* $F(x)$ x $\leftarrow$ y

$F(x) \leftarrow F(y)$

ELSE

put y in CLOSED

ELSE

remove y from OPEN (* each node search one time *)

until T is reached

else T can not reached from S

Starting with the source cell, the process stops if the cell is the last element. If the cell is a subset itself, record it, take it as a subset and puts it neighboring cells on OPEN, as they are treated as the source in this case. Compare F(y) with F(x). Add y to CLOSED if F(x) $\geq$F(y). If F(x)<F(y), put y into OPEN and make F(x)$\leftarrow$F(y). Continue the comparison between F(x) and F(y). When the neighboring cells of the x have all gone through this comparison procedure, take the estimate of their neighbors to do the comparison. Keep on until the largest F(x) is found.

4.2 Estimate Calculation

The A* search algorithm use a heuristic to estimate the distance between the current cell and the target cell. As implemented in the autorouting program, the heuristic is a simple geometric distance approximation. Figure 4.1 illustrates all the possible cell types used to construct a trace, grouped by type.

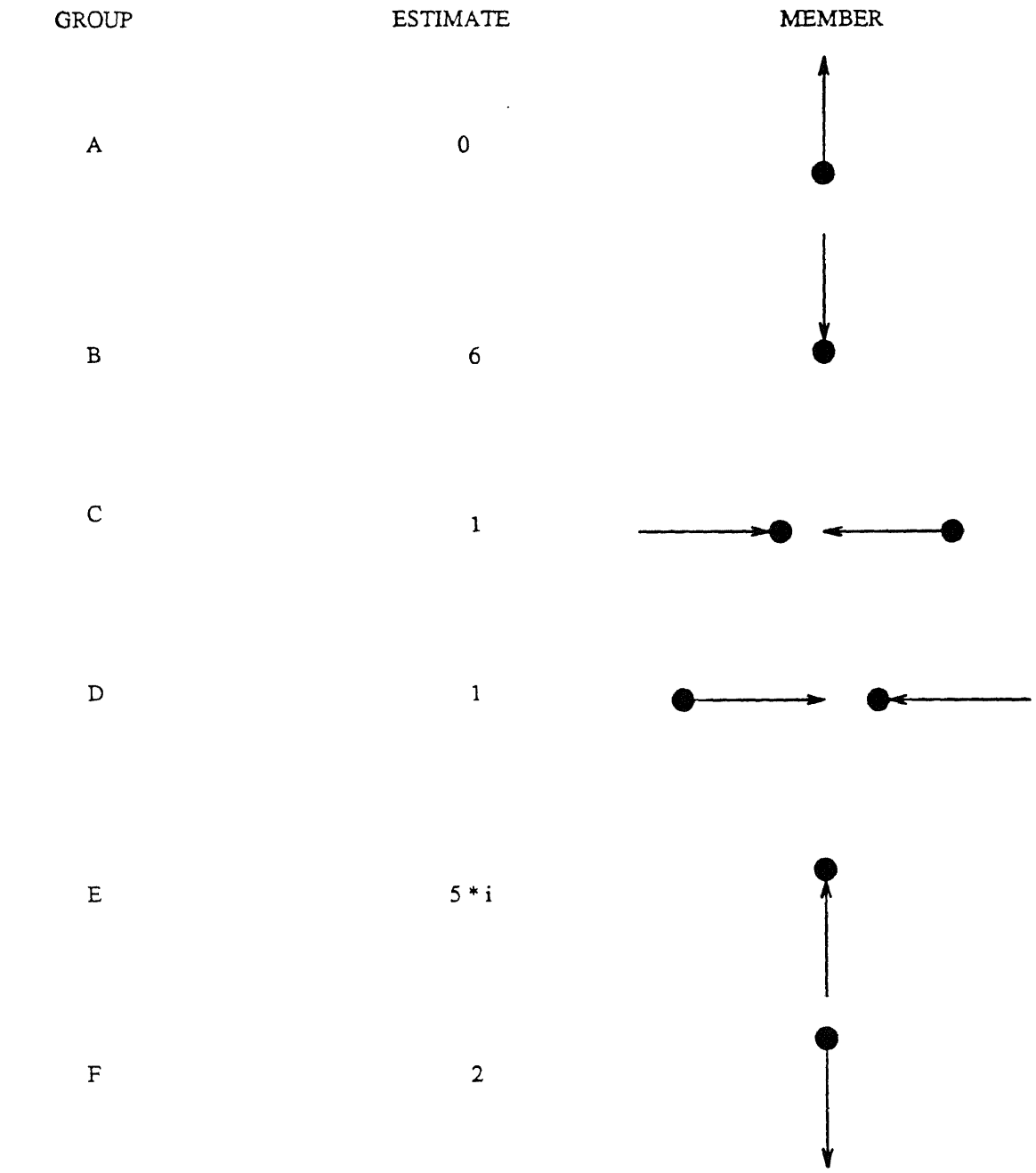
For each group, the estimate of that cell type is also given. These estimates are calculated based on an infinite cell size. If we take the example of Figure 4.1, the estimated of r4c0 equals to

$$2* \text{ group C} + 2 * \text{group D} + 1 * \text{group A}$$
$$= 2 + 2 + 0$$
$$= 4$$

and the estimate of r4c1 equals to

$$1 *\text{group A} + 2 * \text{group C} + 2 * \text{group D} + 1 * \text{group E}$$
$$= 0 + 2 + (5 * 1) + 2$$
$$= 11$$

According to the estimated we can easily find the target cell . The estimate design based on Figure 2.4. Every solvable subset has only one bridge node that has incoming path to this subset. So we design the estimate of bridge node is larger than that of all nodes in the subset that it enters. Every time when we search the bridge node we compare the estimates. Finally, we can get a node with largest estimate that is bridge node and the subset search is finished. The bridge node will be seen as a source cell to searh the other subset. Until OPEN QUEUE is empty, all solvable subsets will be searched.



i: the number of column

Figure 4.1 Definition of Estimate for Type A

Chapter 5
Decomposition Results

The simulation program is written in C code [appendix]. The author tested the program on the lattices of type A, Figure 5.1, on an IBM 286AT Personal Computer with source cell(r4c2). The results are as following:

Source cell : r4c2(14)

6	32	34	34	34	34	36	34	34	32
5	27	29	29	31	29	35	29	29	27
4	22	24	24	30	24	26	24	24	22
3	17	21	19	19	19	25	19	19	17
2	14	20	14	14	14	14	14	14	12
1	13	9	9	9	11	9	9	9	7
0	2	4	4	4	10	4	4	4	2
	0	1	2	3	4	5	6	7	8

$F(X)$

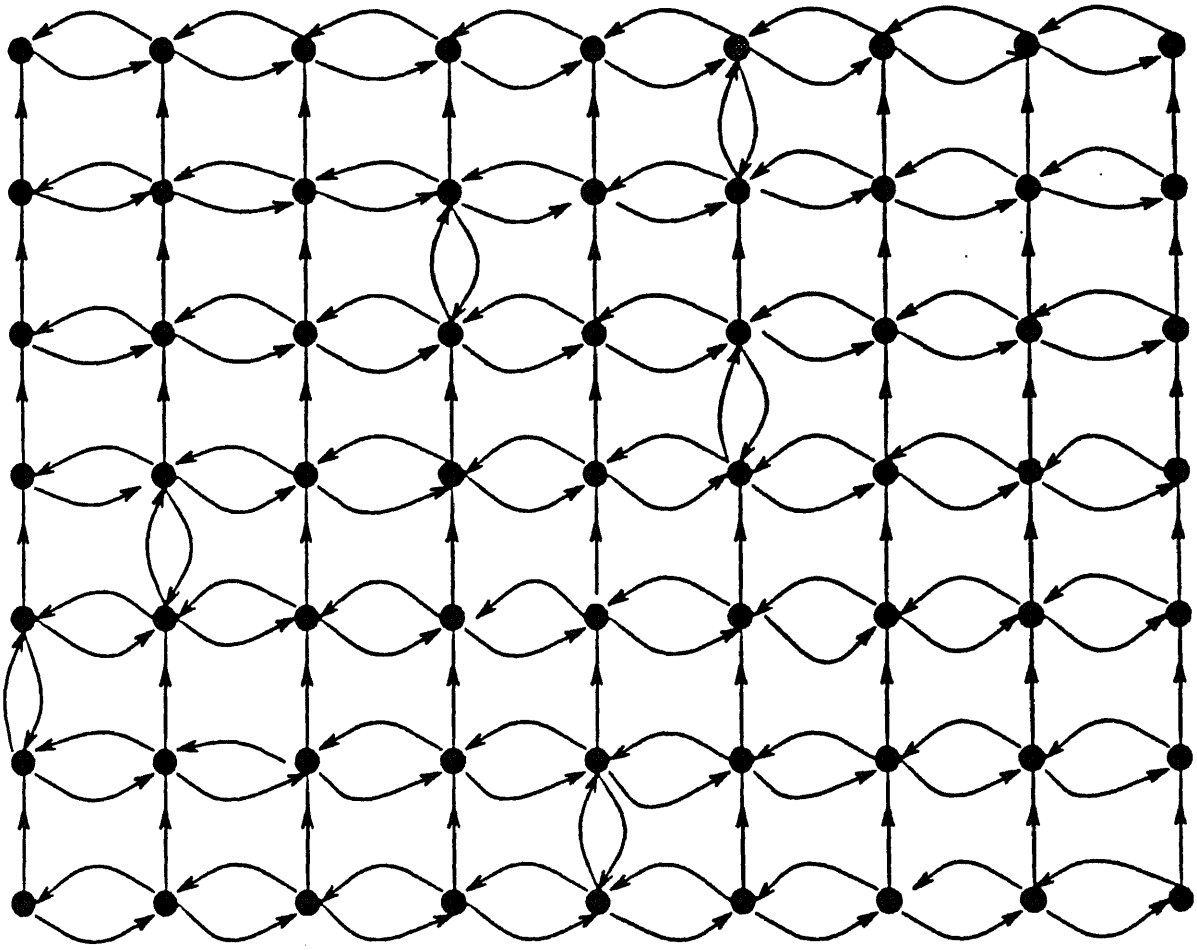


Figure 5.1 Type A Lattice

Source Cell :r4c2(14)

F(x)=14

CLOSED	OPEN
r4c2	r3c2(14)
	r4c1(11)
	r5c2(14)

First Search

Source Cell :r4c2(14)

F(x)=14

CLOSED	OPEN
r4c2	r2c2(14)
r3c2	r3c1(9)
r4c1	r4c0(10)
r5c2	r5c1(9)
	r6c2(14)

Second Search

Source Cell :r4c2(14)

F(x)=14

CLOSED		OPEN
r4c2	r2c2	r1c2(20)
r3c2	r3c1	r2c1(9)
r4c1	r4c0	r3c0(4)
r5c2	r5c1	r5c0(4)
r6c2		r6c1(9)
		r7c2(14)

Third Search

Source Cell :r4c2(14)

F(x)=20

CLOSED			OPEN
r4c2	r2c2	r3c2	r1c2(20)
r3c1	r4c1	r4c0	r1c1(9)
r5c2	r5c1	r6c2	r2c0(4)
r2c1	r3c0	r5c0	r6c0(4)
r6c1	r7c2		r7c1(9)
			r8c2(12)

Fourth Search

Source Cell :r4c2(14)

F(x)=20

CLOSED				OPEN
r4c2	r2c2	r3c2	r1c1	r1c2(20)
r3c1	r4c1	r4c0	r2c0	r0c1(13)
r5c2	r5c1	r6c2	r6c0	r0c0(4)
r2c1	r3c0	r5c0	r8c2	r1c0(4)
r6c1	r7c2	r7c1		r7c0(4)
				r8c0(2)
				r8c1(7)

Fifth Search

Source Cell :r4c2(14)

F(x)=20

CLOSED					OPEN
r4c2	r2c2	r3c2	r1c1	r0c1	r1c2(20)
r3c1	r4c1	r4c0	r2c0	r0c0	
r5c2	r5c1	r6c2	r6c0	r1c0	
r2c1	r3c0	r5c0	r8c2	r7c0	
r6c1	r7c2	r7c1	r7c0	r8c0	
r8c1					

Sixth Search

Now, only one element has the largest estimate in OPEN QUEUE. The search of first subset that is the group of the cells on CLOSED SET is finished. We begin to search the second subset. The element in OPEN will be seen as a source cell for second subset.

Source Cell :r1c2(20)

F(x)=21

CLOSED	OPEN
r1c2	r1c3 (21)

First Search

Now, only one element has the largest estimate in OPEN QUEUE. The search of second subset that is the group of the cells on CLOSED SET is finished. We begin to search the third subset. The element in OPEN will be seen as a source cell for third subset.

Source Cell :r1c3(21)

F(x)=21

CLOSED	OPEN
r1c3	r0c3(17)
	r2c3(19)

First Search

Source Cell :r1c3(21)

F(x)=21

CLOSED		OPEN
r1c3	r0c3	r3c3(19)
r2c3		

Second Search

Source Cell :r1c3(21)

F(x)=21

CLOSED		OPEN
r1c3	r0c3	r4c3(19)
r3c3	r2c3	

Third Search

Source Cell :r1c3(21)

F(x)=25

CLOSED		OPEN
r1c3	r0c3	r5c3(25)
r3c3	r2c3	

Fourth Search

Now, same as before there is only one cell in OPEN QUEUE, the third subset search is finished. We begin to search the fourth subset. The element in OPEN will be seen as a source cell for third subset.

Source Cell :r5c3(25)

F(x)=26

CLOSED	OPEN
r5c3	r5c4(26)
	r6c3(19)

First Search

Source Cell :r5c3(25)

F(x)=26

CLOSED	OPEN
r5c3	r5c4(26)
r6c3	r7c3(19)

Second Search

Source Cell :r5c3(25)

F(x)=26

CLOSED	OPEN
r5c3	r5c4(26)
r6c3	r8c3(17)
r7c3	

Third Search

Source Cell :r5c3(25)

F(x)=26

CLOSED		OPEN
r5c3	r6c3	r5c4(26)
r8c3	r7c3	

Fourth Search

Fourth subset search is finished. Now, start the next subset. The cell r5c4 will be the source cell of the next subset.

Source Cell :r5c4(26)

F(x)=26

CLOSED	OPEN
r5c4	r4c4(24)
	r6c4(24)

First Search

Source Cell :r5c4(26)

F(x)=30

CLOSED		OPEN
r5c3	r5c4	r3c4(30)
r6c3		r7c4(24)

Second Search

Source Cell :r5c4(26)

F(x)=26

CLOSED			OPEN
r5c4	r4c4	r6c4	r3c4(26)
r7c4	r8c4		

Third Search

Fifth subset search is finished. Now, start the next subset. The cell r3c4 will be the source cell of next subset.

Source Cell :r3c4(30)

F(x)=31

CLOSED	OPEN
r3c4	r3c5(31)
	r2c4(24)

First Search

Source Cell :r3c4(30)

F(x)=31

CLOSED		OPEN
r3c4	r2c4	r3c5(31)
		r1c4(24)

Second Search

Source Cell :r3c4(30)

F(x)=31

CLOSED		OPEN
r3c4	r2c4	r3c5(31)
r1c4	r0c4	

Third Search

Sixth subset search is finished. Now, start the next subset. The cell r3c5 will be the source cell of next subset.

Source Cell :r3c5(31)

F(x)=31

CLOSED	OPEN
r3c5	r4c5(31)
	r2c5(29)

First Search

Source Cell :r3c5(31)

F(x)=35

CLOSED		OPEN
r3c5	r2c5	r1c5(29)
r4c5		r5c5(35)

Second Search

Source Cell :r3c5(31)

F(x)=35

CLOSED		OPEN
r0c5	r1c5	r5c5(35)
r2c5	r3c5	
r4c5		

Third Search

Seventh subset search is finished. Now, start the next subset. The cell r5c5 will be the source cell of next subset.

Source Cell :r5c5(35)

F(x)=36

CLOSED	OPEN
r5c5	r5c6(36)
	r6c5(29)

First Search

Source Cell :r5c5(35)

F(x)=36

CLOSED		OPEN
r5c5	r6c5	r5c6(36)
r7c5		

Second Search

Source Cell :r5c5(35)

F(x)=36

CLOSED		OPEN
r5c5	r6c5	r5c6(36)
r7c5	r8c5	

Third Search

Eighth subset search is finished. Now, start the next subset. The cell r5c6 will be the source cell of next subset.

Source Cell :r5c6(36)

F(x)=36

CLOSED	OPEN
r5c6	r4c6(34)
	r6c6(34)

First Search

Source Cell :r5c6(36)

F(x)=36

CLOSED		OPEN
r5c6	r4c6	r3c6(34)
r6c6		r7c6(34)

Second Search

Source Cell :r5c6(36)

F(x)=36

CLOSED		OPEN
r5c6	r4c6	r2c6(34)
r3c6	r6c6	r8c6(32)
r7c6		

Third Search

Source Cell :r5c6(36)

F(x)=36

CLOSED			OPEN
r5c6	r4c6	r2c6	
r3c6	r6c6	r8c6	
r7c6	r1c6	r0c6	

Fourth Search

All solvable subsets are searched. The list follows:

First subset

r4c2 r2c2 r3c2 r1c2 r0c1 r3c1 r4c1 r4c0
r2c0 r0c0 r5c2 r5c1 r6c2 r6c0 r1c0 r2c1
r3c0 r5c0 r8c2 r7c0 r6c1 r7c2 r7c1 r7c0
r8c0 r8c1

Second subset

r1c2

Third subset

r1c3 r0c3 r3c3 r2c3

Fourth subset

r5c3 r6c3 r7c3 r8c3

Fifth subset

r5c4 r4c4 r6c4 r7c4 r8c4

Sixth subset

r3c4 r2c4 r1c4 r0c4

Seventh subset

r3c5 r2c5 r1c5 r0c5 r4c5

Eighth subset

r5c5 r6c5 r7c5 r8c5

Ninth subset

r0c6 r1c6 r2c6 r3c6 r4c6 r5c6 r6c6 r7c6
r8c6

Chapter 6
Discussion and Conclusion

6.1 Discussion

The lattice is represented as a two-dimensional array with one dimension for X axis and the other for y axis in which the dimensions are the number of rows and columns of cells. There are eight possible cell contents, which can be represented with 3 bits per cell. On a matrix of N rows and M columns, we will need N*M*3 bits in total.

The preprocessor structure is also a two-dimensional array, where an entry must be able to represent one of the four compass directions. This takes 2 bits per cell, or N*M*2 bits.

The CLOSED can be represented by a pair of two dimensional, single-bit array , where a bit is one if the corresponding cell has been searched, and zero otherwise . This will take N*M*2 bits in total.

F(x) and G(x) will be similar to the lattice arrays, but they must have a 2 bit integer for each cell, requiring N*M*4 bits in total. Note that if memory usage needs to be minimized at the cost of increased processing time, we could omit the F(x) arrays, and calculate the F values as they are needed from G(x) arrays.

6.2 Conclusion

A decomposition algorithm has been developed to solve the state transition lattice of Type A structure. By dividing the lattice into several solvable subsets which can be solved sequentially, a large amount of computation time can be saved.

References

[1] L. Kleinrock, Qqeueing Systems, Vol.I: Theory, Wiley, 1975.

[2] A.A. Lazar and T.G. Robertazzi, "The Geometry of Markovian Queueing Networks,"Columbia University Technical Report, 1984.

[3] A.A. Lazar and T.G. Robertazzi, "The Geometry of Lattices for Multiclass Markovian Queueing Networks,"*Proc. of the 1984 Conf. on Info. Sci. and Sys.,* Princeton, N.J., pp. 164-168.

[4] A.A. Lazar and J.M. Ferrandiz, "Geometric Analysis of Qqasi-Birth-and-Death Processes by Flow Redirection," *Proc. of the 1987 Conf. on Info. Sci. and Sys.,* The John Hopkins University, Baltimore, MD. March, 1987.

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

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

[7] F. Baskett, K.M. Chandy, R.R. Muntz and F.Palacios," Open, Closed and Mixed Networks of Queues with Different Classes of Customers," *J. ACM, vol. 22, no. 2,* April 1975.

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

[9] Stephen E. Belter, "Computer-aided Routing of Printed Circuit Boards: an Examination of Lee's Algorithm and Possible Enhancements," *BYTE,*(June 1987), 199-208.

[10] Patrick Henry Winston, "Artificial Intelligince", 2nd edition, "Addision-Wesley Publishing Company," 1984.

[11] C.Y. Lee, "An Algorithm for Path Connections and Its Applications,"*IRE Transactions on Electronic Compurers,*" (September 1961),pp 346-365.

[12] Steven L. Tanimoto, The Elements of Artificial Intelligence, (1987, Rockville, Maryland : Computer Science Press),pp 148-164.

Appendix

```
/* ******************************************************** */
/* Program Name : y_0.c                                     */
/* ******************************************************** */
#include <stdio.h>
#include <dos.h>
#include <process.h>
#define point 1
#define xmax 9
#define ymax 7
#define n (xmax*ymax)

struct {
       int x;
       int y;
} buffer[n];
struct neighbor {
       int link;
};
struct {
       struct neighbor north;
       struct neighbor east;
       struct neighbor south;
       struct neighbor west;
} state[xmax][ymax];
int temp[n];
main()
{
unsigned before,after;
int count=0;
int x1,y1;
int num_cont=0;
before=biostime( 0, 0);
read_data();
 clrscr();
 printf("\n\n\n\n\n\n");
 printf("Please Enter Initial State !\n");
  scanf("%d %d",&x1,&y1);
  if(x1<0 || x1>=xmax || y1 <0 || y1 >=ymax)
   {
    printf("Please Enter X From 0 to %d\n",(xmax-1));
    printf("Please Enter Y From 0 to %d\n",(ymax-1));
    printf("Please Try Again !\n");
    scanf("%d %d",&x1,&y1);
    if(x1<0 || x1>=xmax || y1 <0 || y1 >=ymax)
  exit(0);
  }
  while( x1 < xmax && y1 < ymax && x1 >=0 && y1 >= 0)
   {
     switch ((state[x1][y1].north.link + \
              state[x1][y1].east.link  + \
              state[x1][y1].south.link + \
              state[x1][y1].west.link) ) {
            case 0 :
```

```
                    c_comp_0(&x1,&y1,&num_cont,count);
                    break;
               case 1 :
                    c_comp_1(&x1,&y1,&num_cont,count);
                    break;
               case 2 :
                    c_comp_2(&x1,&y1,&num_cont,count);
                    break;
               case 3 :
                    c_comp_3(&x1,&y1,&num_cont,count);
                    break;
               case 4 :
                    c_comp_4(&x1,&y1,&num_cont,count);
                    break;
            }
          count++;
     }
 out_print();
 after=biostime( 0, 0);
 printf("\n\n\n This program use %f second to execute ",
        (unsigned float)(after-before)/18.204);
}
out_print()
{
int i=0;
int k=0;
int j;
FILE *out;
out=fopen("y_0.out","w");
while (i < n ) {
  fprintf(out,"THE GROUP\n");
  for(j=0;j<temp[k];j++)
  {
  fprintf(out,"( %d , %d )\n",buffer[i].x,buffer[i].y);
  i++;
  }
k++;}
fclose(out);
}
read_data()
{
int i,j;
FILE *in;
if((in=fopen("isdn.dat","r")) == NULL)
 {
    puts("Can't open data file !\n");
    exit(0); }
else
 {
    for ( j = 0 ; j < ymax ; j++ )
      for ( i = 0 ; i < xmax ; i++ )
       {
         fscanf( in,"%d %d %d %d",\
           &state[i][j].north.link,\
```

```
            &state[i][j].east.link,\
            &state[i][j].south.link,\
            &state[i][j].west.link);
        }
    }
close(in);
}
c_comp_0(a,b,c,d)
int *a,*b,*c;
int d;
{
buffer[*c].x=*a;
buffer[*c].y=*b;
(*c)++;
(*a)++;
(*b)++;
temp[d]=1;
}
c_comp_1(a,b,c,d)
int *a,*b,*c;
int d;
{
 buffer[*c].x=*a;
 buffer[*c].y=*b;
 (*c)++;
 temp[d]=1;
 if(state[*a][*b].north.link == point )
  {
   (*b)++;
   state[*a][*b].south.link = 0 ;
  }
 else if (state[*a][*b].east.link == point )
  {
  (*a)++;
   state[*a][*b].west.link = 0 ;
   if(*b<(ymax-1))
     state[(*a)-1][(*b)+1].south.link=0;
  }
 else if (state[*a][*b].south.link == point )
  {
  (*b)--;
   state[*a][*b].north.link = 0 ;
  }
 else
  {
  (*a)--;
   state[*a][*b].east.link = 0 ;
   if(*b < (ymax-1))
    state[(*a)+1][(*b)+1].south.link=0;
  }
}
c_comp_4(a,b,c,d)
int *a,*b,*c;
int d;
```

```
{
int i,j;
int sum=0;
*a=*a;
(*b)++;
for(j=0;j<*b;j++)
  for(i=0;i<xmax;i++)
  {
    buffer[*c].x=i;
    buffer[*c].y=j;
    (*c)++;
    sum++;
  }
  for(i=0;i<xmax;i++)
    state[i][*b].south.link=0;
temp[d]=sum;
}



c_comp_2(a,b,c,d)
int *a,*b,*c;
int d;
{
int i,j,_value,t_value;
int sum=0;
if(state[*a][*b].north.link==point)
 {
   if(state[*a][*b].east.link==0)
    for(i=0;i<=*a;i++)
     {
       buffer[*c].x=i;
       buffer[*c].y=*b;
       (*c)++;
       sum++;
       state[i][(*b)+1].south.link=0;
     }
    else
     for(i=*a;i<xmax;i++)
     {
       buffer[*c].x=i;
       buffer[*c].y=*b;
       (*c)++;
       sum++;
       state[i][(*b)+1].south.link=0;
     }
 (*b)++;
 }
else /* state[*a][*b].north.link=0 */
  {
  if(state[*a][*b].south.link==0)
{
 if(*b!=(ymax-1))
 {
```

```
  for(i=0;i<xmax;i++)
   if(state[i][*b].north.link==point)
     {
       _value=i;
       break;
     }

if(_value < *a)
 {
  (*a)--;
state[*a][*b].east.link=0;
     for(i=(*a)+1;i<xmax;i++)
     {
       buffer[*c].x=i;
       buffer[*c].y=*b;
       (*c)++;
       sum++;
       state[i][(*b)+1].south.link=0;
     }
  }
else   /* _value > a */
 {
  (*a)++;

state[*a][*b].west.link=0;
     for(i=0;i<*a;i++)
     {
       buffer[*c].x=i;
       buffer[*c].y=*b;
       (*c)++;
       sum++;
       state[i][(*b)+1].south.link=0;
     }
  }
}
else /*b=ymax*/
  {
   if(*a>xmax/2)
    {
        for(i=*a;i<xmax;i++)
     {
       buffer[*c].x=i;
       buffer[*c].y=*b;
       (*c)++;
       sum++;
     }
   (*a)--;
    state[*a][*b].east.link=0;
    }
 else
   {
     (*a)++;
        for(i=0;i<*a;i++);
     {
```

```
      buffer[*c].x=i;
      buffer[*c].y=*b;
      (*c)++;
      sum++;

    }
  state[*a][*b].west.link=0;
 }
 }
}
else /* south=1 */
    {
    for(j=0;j<*b;j++)
     for(i=0;i<xmax;i++)
      {
      buffer[*c].x=i;
      buffer[*c].y=j;
      (*c)++;
      sum++;
    }
    if((*b)!=(ymax-1))
     {
 for(i=0;i<xmax;i++)
  if(state[i][*b].north.link==point)
    {
      _value=i;
      break;
    }

 for(i=0;i<xmax;i++)
  if(state[i][(*b)-1].north.link==point)
    {
     t_value=i;
      break;
    }

  if(state[*a][*b].east.link==point)
   {
   if( (t_value==(*a)) && (t_value==_value) )
    {
     (*b)++;
     for(i=*a;i<xmax;i++)
     {
      buffer[*c].x=i;
      buffer[*c].y=(*b)-1;
      (*c)++;
      sum++;
      state[i][*b].south.link=0;
     }


    }
   else  if( t_value == *a)
       {
```

```
        state[*a][(*b)+1].south.link=0;
        buffer[*c].x=*a;
        buffer[*c].y=*b;
        (*c)++;
        sum++;
        (*a)++;
        for(i=*a;i<xmax;i++)
         state[i][*b].south.link=0;
         state[*a][*b].west.link=0;
      }



      else if(t_value<_value)
         {
          for(i=*a;i<=t_value;i++)
            {
              buffer[*c].x=i;
              buffer[*c].y=*b;
              (*c)++;
              sum++;
              state[i][(*b)+1].south.link=0;
             }
           *a=t_value+1;
           for(i=*a;i<xmax;i++)
            state[i][*b].south.link=0;
            state[*a][*b].west.link=0;
         }

       else /* t_value >_value */
         {

for(i=*a;i<_value;i++)
{
  buffer[*c].x=i;
  buffer[*c].y=*b;
  (*c)++;
  sum++;
  state[i][(*b)+1].south.link=0;
}
for(i=_value+1;i<xmax;i++)
{
  buffer[*c].x=i;
  buffer[*c].y=*b;
  (*c)++;
  sum++;
  state[i][(*b)+1].south.link=0;
}
  *a=_value;
  state[*a][*b].south.link=0;
  state[*a][*b].east.link=0;
  state[*a][*b].west.link=0;
  }
```

```
 }
else  /* east=0 */
 {
  if((t_value==*a) && (t_value==_value))
   {
    for(i=0;i<=(*a);i++)
     {
        buffer[*c].x=i;
        buffer[*c].y=*b;
        (*c)++;
        sum++;
        state[i][(*b)+1].south.link=0;
     }
     (*b)++;
   }
   else if(t_value == *a)
      {
        state[*a][(*b)+1].south.link=0;
        buffer[*c].x=*a;
        buffer[*c].y=*b;
        (*c)++;
        sum++;
        (*a)--;
        for(i=0;i<=*a;i++)
         state[i][*b].south.link=0;
        state[*a][*b].east.link=0;
      }
     if(t_value>_value)
       {
        for(i=t_value;i<=*a;i++)
          {
            buffer[*c].x=i;
            buffer[*c].y=*b;
            (*c)++;
            sum++;
            state[i][(*b)+1].south.link=0;
           }
         for(i=0;i<t_value;i++)
          state[i][*b].south.link=0;
         *a=_value-1;
          state[*a][*b].east.link=0;
       }

     else /* <= */
       {
for(i=0;i<= *a;i++)
 if(i != _value)
{
  buffer[*c].x=i;
  buffer[*c].y=*b;
  (*c)++;
  sum++;
  state[i][(*b)+1].south.link=0;
}
```

```
      *a=_value;
     state[*a][*b].west.link=0;
     state[*a][*b].south.link=0;
     state[*a][*b].east.link=0;
     }
}
 }
 else /* b= ymax */

 {
 for(i=0;i<xmax;i++)
   if(state[i][(*b)-1].north.link==point)
    {
      t_value=i;
      break;
    }
    if(state[*a][*b].east.link==point)
    {
    if(t_value  <= (*a))
    {
    for(i=0;i<=*a;i++)
     {
       buffer[*c].x=i;
       buffer[*c].y=*b;
       (*c)++;
       sum++;
     }
     (*a)++;
     for(i=*a;i<xmax;i++)
     state[i][*b].south.link=0;
     state[*a][*b].west.link=0;
     }
    else
     for(i=*a;i<xmax;i++)
     {
      buffer[*c].x=i;
      buffer[*c].y=*b;
      (*c)++;
      sum++;
     }
     (*a)--;
     for(i=0 ;i<=*a;i++)
     state[i][*b].south.link=0;
     state[*a][*b].east.link=0;
     }
/*      }  */
    else
    {
     for(i=0;i<xmax;i++)
      {
       buffer[*c].x=i;
       buffer[*c].y=*b;
       (*c)++;
       sum++;
```

```
      }
      (*a)++;
      (*b)++;
    }
 }
 }
temp[d]=sum;
}
c_comp_3(a,b,c,d)
int *a,*b,*c;
int d;
{
int i,j,_valud,_valup;
int sum=0;
if(state[*a][*b].north.link==point)
  {
    (*b)++;
   if(state[*a][(*b)-1].south.link==point)
   {
    for(j=0;j<=*b;j++)
     for(i=0;i<xmax;i++)
      if(j == *b )
       state[i][*b].south.link=0;
     else
      {
        buffer[*c].x=i;
        buffer[*c].y=j;
        (*c)++;
        sum++;
      }
  }
  else  /* state[*a][*b].south.link != point */
    {

      for(i=0;i<xmax;i++)
        {
          buffer[*c].x=i;
          buffer[*c].y=(*b)-1;
          (*c)++;
          sum++;
          state[i][*b].south.link=0;
        }
    }
 }
else   /*north=0 */
  {
    if((*b) < (ymax-1))
     {
       for(j=0;j<=(*b)-1;j++)
        for(i=0;i<xmax;i++)
         {
           buffer[*c].x=i;
           buffer[*c].y=j;
           (*c)++;
```

```
    sum++;
  }
 for(i=0;i<xmax;i++)
  if(state[i][*b].north.link==point)
  {
  _valup=i;
  break;
  }
 for(i=0;i<xmax;i++)
  if(state[i][(*b)-1].north.link==point)
  {
  _valud=i;
  break;
  }

   if(_valup < *a )
    {
     if(_valud >= *a )
     {
     for(i=*a;i<xmax;i++)
      {

        state[i][(*b)+1].south.link=0;
        buffer[*c].x=i;
        buffer[*c].y=*b;
        (*c)++;
        sum++;
      }
      for(i=0;i<*a;i++)
       state[i][*b].south.link=0;
       (*a)--;

       state[*a][*b].east.link=0;
    }
  else if ( _valud > _valup)
    {
     for(i=_valud+1 ; i < xmax ; i++)
       {
         state[i][(*b)+1].south.link=0;
         buffer[*c].x=i;
         buffer[*c].y=*b;
         (*c)++;
         sum++;
       }
      *a = _valud;
      state[*a][*b].east.link=0;
      for(i=0 ;i<=*a ;i++)
      state[i][*b].south.link=0;
    }
else /* _valud <= _valup */
  {
   *a = _valup;
   for(i=0;i<xmax;i++)
     if( i != *a )
```

```
            {
              state[i][(*b)+1].south.link=0;
              buffer[*c].x=i;
              buffer[*c].y=*b;
              (*c)++;
              sum++;
            }
             state[*a][*b].east.link=0;
             state[*a][*b].south.link=0;
             state[*a][*b].west.link=0;
         }
       }
else  /* _valup > a */
 {
   if(_valud<=*a)
    {
    for(i=0 ;i<= *a;i++)
     {

       state[i][(*b)+1].south.link=0;
       buffer[*c].x=i;
       buffer[*c].y=*b;
       (*c)++;
       sum++;
     }
      (*a)++;
     for(i=*a;i<xmax;i++)
      state[i][*b].south.link=0;

      state[*a][*b].west.link=0;
   }
  else if ( _valud < _valup )
   {
     *a= _valud ;
     for(i=0 ;i<*a ;i++)
      {

        state[i][(*b)+1].south.link=0;
        buffer[*c].x=i;
        buffer[*c].y=*b;
        (*c)++;
        sum++;
      }
      for(i=*a;i<xmax;i++)
       state[i][*b].south.link=0;
       state[*a][*b].west.link=0;
   }
  else /* _valud >= _valup */
   {
      *a=_valup;
      for(i=0;i<xmax;i++)
        if( i != *a )
          {
            state[i][(*b)+1].south.link=0;
```

```
                buffer[*c].x=i;
                buffer[*c].y=*b;
                (*c)++;
                sum++;
              }
               state[*a][*b].east.link=0;
               state[*a][*b].west.link=0;
               state[*a][*b].south.link=0;
    }
 }
}
else  /* b=ymax */
 {
     for(j=0;j< (*b) ;j++)
      for(i=0;i<xmax;i++)
       {
         buffer[*c].x=i;
         buffer[*c].y=j;
         (*c)++;
         sum++;
       }
      for(i=0;i<xmax;i++)
       if(state[i][(*b)-1].north.link==point)
        {
          _valup = i ;
          break;
        }
    if( _valup < (*a) )
      {
        (*a)++;
        for(i=0 ; i< xmax; i++)
        {
         if(i >= (*a) )
             state[i][*b].south.link=0;
         else
           {
             buffer[*c].x=i;
             buffer[*c].y=*b;
             (*c)++;
             sum++;
           }
        }
       state[*a][*b].west.link=0;
      }
    else if(_valup ==(*a))
      {
        if(*a > xmax/2)
        {
        (*a)--;
        for(i=0 ; i< xmax; i++)
        {
         if(i <= *a )
             state[i][*b].south.link=0;
         else
```

```
                {
                  buffer[*c].x=i;
                  buffer[*c].y=*b;
                  (*c)++;
                  sum++;
                }
              }
             state[*a][*b].east.link=0;
           }
         else /* a <xmax/2 */
          {
           (*a)++;
           for(i=0 ; i< xmax; i++)
           {
            if(i >=(*a))
                state[i][*b].south.link=0;
            else
              {
                buffer[*c].x=i;
                buffer[*c].y=*b;
                (*c)++;
                sum++;
              }
             }
           state[*a][*b].west.link=0;
          }
      }

    else /* _valup > a */
      {
       (*a)--;
       for(i=0 ; i< xmax; i++)
        if(i <= *a )
            state[i][*b].south.link=0;
        else
          {
            buffer[*c].x=i;
            buffer[*c].y=*b;
            (*c)++;
            sum++;
          }
        state[*a][*b].east.link=0;
       }
  }
}
temp[d]=sum;
}
```