NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

A Knowledge Acquisition Method: Transformation of Algorithms and Programs with infoMaps (TAPi)

Thompson Cummings

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfilment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

May 1991

ISBN 0-315-68766-5

Abstract

A Knowledge Acquisition Method: Transformation of Algorithms and Programs with infoMaps (TAPi)

Thompson Cummings

This thesis proposes a methodology for the transformation of algorithms and programs. Conventional programming languages and development tools do not provide for the proper writing and documentation of algorithms and programs. Thus, this methodology which is unconventional, yet easy to learn and efficient in practice, is an attempt to correct such inadequacies of conventional tools.

A developer or user using this methodology for the transformation of algorithms and programs would be allowed to make several discoveries. For example, he/she would be able to find out if the existing algorithms and programs are built out of reusable components. Apart from the number of statements in an algorithm or program, other knowledge such as transitions and states become readily available; in conventional representation they are not. This methodology allows the user or developer to document various components of algorithms and programs and helps him/her to communicate his/her understanding of them. Our methodology works well in practice and therefore is recommended as a knowledge acquisition tool.

Acknowledgements

I would like to thank my thesis supervisor, Dr. W.M. Jaworski, for his support and guidance throughout this project. He was readily available for consultation, and the time and effort he put into guiding me are greatly appreciated. I would like, also, to thank both Dr. L. Tao and Dr. R. Shinghal for providing some source algorithms. The time Dr. Tao made himself available for consultation is very much appreciated.

Many thanks to S. Cattou, B. Dash, A. Wongyai, M. Wallace and many others for their useful comments. Many thanks also to Mrs. L. Vadzis of AUCC for the tremendous support given to me throughout the project.

Table of Contents

ABBREVIATIONS

Abbreviation	Meaning
1st	first
BM	Boyer-Moore
corresp	corresponding
e.g.	for example
elem	element
fig.	figure
gt	greater than
i.e.	that is
KMP	Knuth-Morris-Pratt
lt	less than
max	maximum
min	mimimum
no(s).	number(s)
pat	pattern
pos	position
pt	point
ptr	pointer
QS	Quick Search
resp	respectively
SF	Straight Forward
soln	solution
SS	Substring Search
TAPi	Transformation of Algorithms and Programs with infoMaps
temp	temperature
txt	text
wt(s)	weight(s)

List of Figures

List of Figures (cont.)

List of Figures (cont.)

Figures of Appendix I

Figures of Appendix II

Figures of Appendix III

Figures of Appendix IV

Figures of Appendix V

Chapter 1

Introduction

1.1 Purpose:

This thesis is aimed at providing a methodology for enabling Software Engineers tc re-engineer software products in general. However, its objective is to specifically provide for such transformation of algorithms and programs that is easy to follow, processable and highly expressive. The methodology proposed in this thesis is named **T**ransformation of **A**lgorithms and **P**rograms with **i**nfoMaps(**TAPi**).

The purpose of TAPi is threefold. Firstly, by using this methodology the user would discover if the existing algorithms are built out of reusable components. In such a case, there is no need to build new algorithm(s) from scratch. Secondly, TAPi provides the means whereby the algorithm(s) can be documented properly which is crucial for its maintenance - for systems of the real world do not remain static. Thirdly, it provides users with more meaningful knowledge about the algorithm(s). This means that one is now able to look at an algorithm not only as containing a sequence of statements but also, as a structure built out of reusable components.

The structuring of the knowledge should be consistent and should capture the crucial aspects of the algorithm or program - data model, data flow and control flow.

1.2 State of Art:

Most computer languages are poor vehicles for modeling algorithms because they force the specification of implementation details that are irrelevant to the algorithm [44]. A model that contains unnecessary details would limit the choice of design decisions and divert attention from the real issues.

1.3 TAPi methodology:

TAPi methodology is a model-building process which involves the recovery of the following:

a) hierarchical model;

b) data model;

c) data flow;

d) control flow.

Such models provide the format for representing the knowledge as it is being recovered.

1.4 Hierarchical model:

In the hierarchical model, modules are identified and then related as a hierarchy. The hierarchy is referred to as a tree of calls if there exists a structure in which a main algorithm or program calls other modules which, in turn, call other modules.

1.5 Data model:

In the data model, attributes (or data-objects) are identified and then related to the algorithms (or objects). Each attribute type is also identified in the process [13,23,24].

1.6 Data flow:

In this model, two structures are used for representing data flows. One structure represents the flow of data-objects amongst the various algorithms (e.g Fig. 5.3.2.2) and the other the flow of data-objects for each algorithm (e.g Fig. 5.2.3.3). In the latter, the flow is local to the algorithm [13,23,24,28..31].

1.7 Control flow:

The control flow model represents the temporal, behavioral and "control" aspects of an algorithm or program. Two structures are presented for the flow. One structure relates the components of algorithms and would be used to represent and transform the "control" aspects. The other structure relates paths to transitions and is to used for verifying the transformed algorithms.

1.8 Inheritance Tracing:

Inheritance may be defined as the sharing of attributes and components between algorithms. The components of algorithms are examined in order to trace inheritance. The components are also examined to established their reusability.

1.9 Reusability:

When code is developed in a conventional fashion, a variety of requirements and design structures are mixed together, so that their individual structures may not be at all apparent. Therefore, a representation of the design must allow the designer to edit individual factors and to combine several factors, obtaining a new component [5].

For a component to be reusable, it and anything that it relies upon (its context) must be known. This requires an understanding of what the component does, and where to find it and its context.

1.10 Expressiveness:

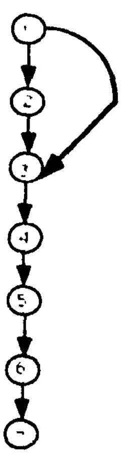
Many computer languages or development tools are not highly expressive [50] and thus, limit one's ability to write well documented algorithms. The development environment used in this research tries to overcome the problem by providing an effective notational technology.

1.11 Organization of the Thesis:

The thesis consists of seven chapters and five appendices. In chapter 2, a new notational technology, infoMaps, is introduced and explained with an example. The TAPi methodology is outlined and discussed in chapter 3. Chapter 4 deals with program normalization and optimization. The detailed steps of algorithm processing are demonstrated on several algorithms in chapter 5.

The required properties of an enhanced TAPi environment are stated and discussed in chapter 6. The thesis concludes, in chapter 7, with discussions on achievements, limitations and future research suggestions. Appendix I gives the syntax of infoMaps notation. Appendix II shows the normalization and optimization of the All-zero row program. Appendices III, IV and V give further transformation of the algorithms from chapter 5.

A graph representing a suggested reading sequence for this thesis is shown in Fig. 1.11.1 (a). The circles of the graph represent the various chapters of this thesis. The graph transformation into an infoMap is shown in Fig. 1.11.1 (b). The infoMap representation is explained in chapter 2. As shown in the infoMap, a reader who is "New to infoMaps" should read the chapters in the sequence indicated (the outlined area). A reader who is "Familiar with infoMaps" would skip chapter 2.



A	A	*[View]*
V	.	New to infoMaps
.	V	Familiar with infoMaps
S	S	*[Chapter]*
1	1	Introduction
2	.	infoMaps
3	2	The TAPi methodology
4	3	Program Normalization and Optimization
5	4	Application of TAPi
6	5	Required Properties of the TAPi Environment
7	6	Conclusions

a) A conventional representation **b) infoMap**

Fig. 1.11.1 Reading Sequence

Chapter 2

infoMaps

2.1 Introduction:

A spreadsheet-based modeling tool, infoMaps [25,28..31] is used for representation and rewriting of algorithms and programs . The infoMaps may be viewed as a collection of sets and their relationships. This modeling tool is illustrated and explained in sections 2.2 and 2.3.

2.2 infoSchemas:

A library of **infoSchemas** prescribes the relationships for the generic objects like control flow graph, data flow graph, data model, sequential program, object-oriented program and other concepts from software engineering or other disciplines. For instance, the infoSchema in Fig. 2.2.1 defines the generic structure of any infoMap representing sequential program.

O	*{Transition}*
L	*{State}*
G	*{preCondition}*
S	*{Action}*
G	*{postCondition}*

Fig. 2.2.1 Generic infoSchema for a sequential program

The entries "**O**", "**L**", "**G**", "**S**" and "**G**" allocate the specific roles respectively to the sets **{Transition}**, **{State}**, **{preCondition}**, **{Action}** and

{postCondition}. The meaning of the entries is: <u>each</u> transition <u>links</u> states; each transition is implemented by a <u>sequence</u> of actions <u>guarded</u> by preconditions and <u>asserted</u> by postconditions. The specific role allocated to a set prescribes a limited number of roles/entries permitted for the members of the set. For instance for a set in role **G** , valid entries for the set members are: **t**-rue, **f**-alse and **c**-omplementary.

2.3 An infoMap example:

In order to explain infoMaps as an algorithm or program representation and processing tool, the problem of a binary search in an ordered array [32,35] is considered. The Pascal-type solution (see Fig. 2.3.1) is normalized, i.e. translated "as-is", into an infoMap.

```
1 :   found := false;
2 :   while(first <= last)
      and not found do begin
             i := (first + last) div 2;
3 :          if a[i]<X then first := i+1
4 :          elseif a[i]>X then last := i-1
             else found := true
      end;
5 :   if a[i] = X then
             X found at i
      else
             X not found;
6 :
```

a) Pascal-type code

	1	2	3	4	5	6	7	8	9	10	11	12
1	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	9	*{Transition}*	
2	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	6	*{State}*	
3	s	.	.	.	.	.	.	.	.			1:.
4	d	s	s	d	.	d	d	.	.			2:.
5	.	d	.	s	s	.	.	.	.			3:.
6	.	.	.	.	d	s	s	.	.			4:.
7	.	.	d	.	.	.	.	s	s			5:.
8	.	.	.	.	.	.	.	d	d			6:.
9	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	4	*{preCondition}*	
10	.	t	f	.	.	.	.	.	.			first<=last and not found
11	.	.	.	t	f	.	.	.	.			a[i]<X
12	.	.	.	.	.	t	f	.	.			a[i]>X
13	.	.	.	.	.	.	.	t	f			a[i]=X
14	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	7	*{Action}*	
15	1	.	.	.	.	.	.	.	.			found:=false
16	.	1	.	.	.	.	.	.	.			i:=(first+last) div 2
17	.	.	.	1	.	.	.	.	.			first:= i+1
18	.	.	.	.	.	1	.	.	.			last:=i-1
19	.	.	.	.	.	.	1	.	.			found:+true
20	.	.	.	.	.	.	.	1	.			X found at i
21	.	.	.	.	.	.	.	.	1			X not found
22												
23												

b) Pascal code rewritten as infoMap

Fig. 2.3.1 Binary Search program

Cells in rows 3..8 and columns 1..9 specify control flow graph component of the normalized solution. Entries in the cells of individual columns specify individual transitions connecting the rows i.e. states of the graph. The character "**s**" means **s**ource and "**d**" **d**estination. The cell entries in rows 10..13 and columns 1..9 specify guard part and the cell entries in rows 15..21 and columns 1..9 command part of the Guarded Commands [18].

The cell entries in rows 3..8 and column 12 show the states of the control flow graph to which textual descriptions could be added.

Chapter 3

The TAPi Methodology

3.1 Introduction:

The TAPi methodology is a model-building process which captures the syntax and semantics of an algorithm or program. A model is an abstraction of reality for the purpose of understanding it before building it. A model omits non-essential details, therefore it is easier to manipulate the model than the original entity [44]. The models built in TAPi use precise notations and are verified to ensure that the requirements of the algorithms or programs are satisfied. The four models partition an algorithm into views that can be represented and manipulated. Each model can be examined and understood independently.

The hierarchical model represents the hierarchical structure of objects. The data model represents the static, structural, "data" aspects of an algorithm. The control flow model represents the temporal, behavioral, "control" aspects of an algorithm. The data flow model represents the transformational, "function" aspects of an algorithm. A typical algorithm uses data structures (data model), sequences actions in time (control flow) and transforms values (data flow). Each model contains references to components in other models.

3.2 Structuring selected views:

The views mentioned above are structured as shown in Fig. 3.2.1. Each view is represented as a model for transforming algorithms or programs. The first column is used to list, under the appropriate set, the data-objects and code used in the algorithm. The second column shows that both the algorithms (i.e **{Algorithm}**) and the procedures (i.e **{Procedure}**) are represented as hierarchies marked with "**H**". The "x" indicates the cardinality of the sets, which is unknown at this stage.

	A	*A*	*A*	*A*	*A*	*A*	*A*	4	*{View}*
	v	.	.	.	.	.	.		1:. Hierarchy
	.	v	v	.	.	.	.		2:. Data Model
	.	.	.	v	v	.	.		3:. Data Flow
	.	.	.	.	.	v	v		4:. Control Flow
	H	*O*	.	*O*	.	.	.	x	*{Algorithm}*
{Declarative Statement}	.	.	*O*	.	.	.	.	x	*{Type}*
{Declarative Statement}	.	*M*	*M*	*F*	*O*	.	.	x	*{Data-object/Attribute}*
	.	.	.	.	.	*O*	.	x	*{Path}*
	.	.	.	.	*F*	*S*	*O*	x	*{Transition}*
	.	.	.	.	*F*	.	*L*	x	*{State}*
{Conditional Statement}	.	.	.	.	*F*	.	*G*	x	*{preCondition}*
{Procedural Statement}	*H*	.	.	.	*F*	.	*S*	x	*{Procedure}*
{Imperative Statement}	.	.	.	.	*F*	.	*S*	x	*{Action}*
{Conditional Statement}	.	.	.	.	*F*	.	*G*	x	*{postCondition}*

Fig. 3.2.1 Selected views used by TAPi

The third column shows a **O**-ne to **M**-any relationship between algorithm and data-objects. There is also, shown by fourth column, a **O**-ne to **M**-any relationship between type and data-objects i.e. for a given type there are many data-objects. "**F**" entry in fifth column indicates the flow of data-objects to and from an algorithm.

The steps used by TAPi for modeling sequential algorithms are presented in Fig. 3.2.2.

```
O O O O O O O O O O O O  12 {Transition}
o . . . . . . . . . . .     system's overview accepted
. o . . . . . . . . . .     system's overview rejected
. . o . . . . . . . . .     hierarchical model accepted
. . . o . . . . . . . .     hierarchical model rejected
. . . . o . . . . . . .     data model accepted
. . . . . o . . . . . .     data model rejected
. . . . . . o . . . . .     data flow accepted
. . . . . . . o . . . .     data flow rejected
. . . . . . . . o . . .     control flow accepted
. . . . . . . . . o . .     control flow rejected
. . . . . . . . . . o .     product accepted
. . . . . . . . . . . o     product rejected
L L L L L L L L L L L L  8 {State}
s s . . . . . . . . . .     1:.Start System's overview
d . s s . . . . . . . .     2:.Building Hierarchical Model
. . d . s s . . . . . .     3:.Building Data Model
. . . . d . s s . . . .     4:.Building Data Flow model
. . . . . . d . s s . .     5:.Building Control flow model
. . . . . . . . d . s s     6:.Testing product
. d . d . d . d . d . d     7:.Select an appropriate state
. . . . . . . . . . d .     8:.Exit
G G G G G G G G G G G G  6 {preCondition}
t f . . . . . . . . . .     System's overview specified
. . t f . . . . . . . .     Hierarchical model completed
. . . . t f . . . . . .     Data Model completed
. . . . . . t f . . . .     Data flow model completed
. . . . . . . . t f . .     Control flow completed
. . . . . . . . . . t f     Product test completed
S S S S S S S S S S S S  6 {Action}
1 . . . . . . . . . . .     Build infoSchema from the Knowledge source
. . 1 . . . . . . . . .     Build Hierarchical model
. . . . 1 . . . . . . .     Build Data model i.e. identify objects and their attributes
. . . . . . 1 . . . . .     Build a Data flow model
. . . . . . . . 1 . . .     Build Control Flow model
. . . . . . . . . . 1 .     Test product i.e Trace control flow
G G G G G G G G G G G G  . {postCondition}
```

Fig. 3.2.2 A knowledge acquisition process

Fig. 3.2.2 shows that at a given state it may be required to select another state and return afterwards to have it completed.

Some TApi steps are explained and discussed below using a Straight Forward search algorithm presented in Fig. 3.2.3.

```
1.   i := 0;
2.   while(i<n-m+1) do
3.        i := i+1
4.        j := i; k := 1;
5.        while(PAT(k) = TXT(j)) do
6.             if(k = m) then
7.                  return("pattern found at", i)
               else do
8.                  j := j+1;
9.                  k := k+1;
               end;
          end;
     end;
10.  return("pattern not found");
```

Fig. 3.2.3 A Straight Forward search algorithm

3.3 Hierarchical view:

The first step of the methodology is to consider if the hierarchical model is required. It is easier to understand a program or algorithm if it is broken into modules. This, in turn, facilitates both debugging and reusability. If there are many modules then a hierarchical structure of the module dependency must be explicit, so that whenever one module invokes another, the latter must be explicitly imported to the former. The infoSchema for this model is extracted from infoSchema in Fig. 3.2.1 and given in Fig. 3.3.1.

```
                        A  4  (View)
                        v      1:. Hierarchy
                        H  x  (Algorithm)
(Procedural Statement)  H  x  (Procedure)
```

Fig. 3.3.1 Hierarchical view used by TAPi

No decomposition is required for the algorithm in Fig. 3.2.3 and therefore, a hierarchical model is not needed.

3.4 Data model view:

The second step in the methodology is to recover the data model. Fig. 3.4.1 gives the infoSchema for such model. Objects, attributes and types of attributes are identified. It is imperative at this stage to give meaning to objects and attributes, which would be used when building the data flow and control flow.

	A	*A*	4	*{View}*
	v	v		2:. Data Model
	O	.	x	*{Algorithm}*
{Declarative Statement}	.	*O*	x	*{Type}*
{Declarative Statement}	*M*	*M*	x	*{Data-object/Attribute}*

Fig. 3.4.1 Data model view used by TAPi

The data model view for the Straight Forward search is shown in Fig.3.4.2. In the first column the data-objects are declared and corresponding textual descriptions are given in column five. The character "**a**" means attribute. For instance the data-object "text length" is an attribute and is of type integer.

	O	.	.	1	*{Algorithm}*
	o	.	.	.	Straight Forward
{Declarative Statement}	.	*O*	*O*	2	*{Type}*
character	.	o	.	.	character
integer	.	.	o	.	integer
{Declarative Statement}	*M*	*M*	*M*	9	*{Data-object/Attribute}*
i	a	.	v		guess ptr. for pat within the text
n	a	.	v		length of text string
m	a	.	v		length of pattern substring
PAT[]	a	v	.		pattern substring
TXT[]	a	v	.		text string
k	a	.	v		subscript indicating current pos in pattern
j	a	.	v		subscript indicating current pos in text
"pattern found at"	a	.	v		success message
"pattern not found"	a	.	v		failure message

Fig. 3.4.2 Data model of a Straight Forward search algorithm

3.5 Data flow view:

Step 3 is to construct the data flow model. Fig. 3.5.1 gives the infoSchema for building the model. Two structures are used for building the model. One structure represents the flow of data-objects between modules and the other the flow of data-objects within each module.

	A	A	4	*{View}*
	v	v		3:. Data Flow
	O	.	x	*{Algorithm}*
{Declarative Statement}	F	O	x	*{Data-object/Attribute}*
	.	F	x	*{Transition}*
	.	F	x	*{State}*
{Conditional Statement}	.	F	x	*{preCondition}*
{Procedural Statement}	.	F	x	*{Procedure}*
{Imperative Statement}	.	F	x	*{Action}*
{Conditional Statement}	.	F	x	*{postCondition}*

Fig. 3.5.1 Data flow view used by TAPi

The data flow of a module is demonstrated in Fig. 3.5.2. There are 7 data-objects of which 4 flow in, 2 flow out and 1 flows both in and out. These are indicated by "i", "o" and "b" respectively in the second column.

	O	1	*{Algorithm}*
	o	.	Straight Forward
{Declarative Statement}	F	7	*{Data-object/Attribute}*
i	b		guess ptr. for pat within the text
n	i		length of text string
m	i		length of pattern substring
PAT[]	i		pattern substring
TXT[]	i		text string
"pattern found at"	o		success message
"pattern not found"	o		failure message

Fig. 3.5.2 Data flow of a Straight Forward search algorithm

3.6 Control flow view:

The fourth model to be recovered is the control flow. The infoSchema for the model is given in Fig. 3.6.1. In this model the "control" aspect of the algorithms is transformed into infoMaps.

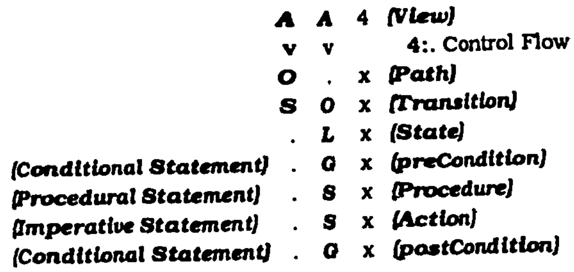


Fig. 3.6.1 Control flow view used by TAPi

This transformation process consists of nine basic steps. These steps are not illustrated in this section because to fully understand this process the reader is required to read chapter 4. The steps are illustrated in chapter 5 and Appendices III-V. In general the steps are as follows:

a) presenting original source code listings of the algorithm;

b) editing source code (listings) of the algorithm by identifying and adding states i.e. identifying decision points or major activities in an algorithm/ program;

c) normalizing the algorithm's code by transforming it, "as is", into a normalized infoMap;

d) adding descriptions to the states and transitions of normalized infoMap produced in c);

e) showing normalized state diagram i.e. infoMap with states and transitions only;

f) indicating area of normalized infoMap to be optimized i.e. indicating transitions, states that can be merged and redundant conditions, procedures and actions that can be eliminated;

g) Optimizing and documenting infoMap;

h) representing optimized state diagram as Structured English;

i) generating source code.

The transformed algorithm/ program is verified using paths. These paths are of the control flow model. Fig. 3.6.1 shows that a path is described as a sequence of transitions. Sometimes, identifying all the paths could be a tedious process. The verification technique used here is demonstrated in fig. 6.2.1.1 to Fig. 6.2.1.3.

The structures of the various models provide for both code and description insertion into their models, which is key to good documentation.

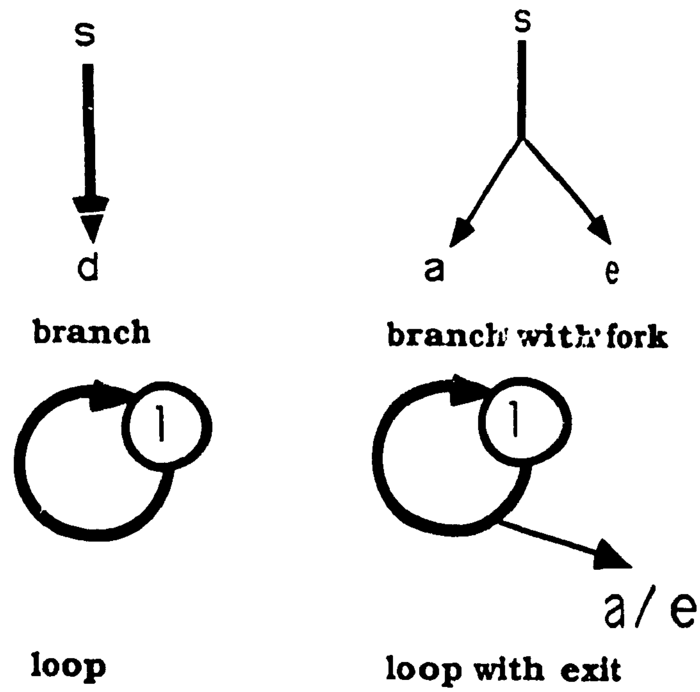
Chapter 4

Program Normalization and Optimization

4.1 Introduction

Programming examples are used in this chapter to further explain and demonstrate the power and simplicity of infoMaps. Small programs generated from infoMap based specifications are included. These programs contain conventional control structures. Properties of elementary control structures are well researched and discussed in many publications describing alternative control structures and programming styles [2,18,20,32..35,43].

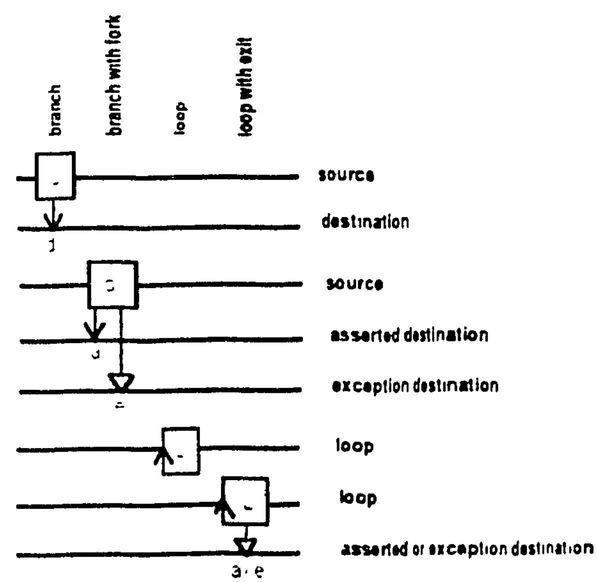
This chapter comprises of five sections. In Section 4.2 the representations of control flow graphs and their implementations in Pascal, Pancode, EPN and infoMaps are compared. Control flow graph of the control structures discussed in [2,18,20,32..35,43] can be represented with 4 elementary graphs (see Fig. 4.1.2) or their composition. Those generic graphs namely Sequence, Sequence with assertion or exception, Loop and Loop with exit were chosen to allow modeling and manipulation of control flow graphs and programs within the spreadsheet environment of infoMaps.



(a) graphical

```
O  O  O  O  O  {Generic Graph}
o  .  .  .  .      branch from s to d
.  o  .  .  .      branch from s to either a or e
.  .  o  .  .      loop
.  .  .  o  .      loop with exit to e
.  .  .  .  o      loop with exit to a
L  L  L  L  L  {Node/State}
s  .  .  .  .      source
d  .  .  .  .      destination
.  s  .  .  .      source
.  a  .  .  .      asserted destination
.  e  .  .  .      exeception destination
.  .  l  .  .      loop
.  .  .  l  .      loop
.  .  .  e  .      exception destination
.  .  .  .  l      loop
.     .  .  a      asserted destination
```

(b) infoMap



(c) schematic

Fig 4.1.2 Generic control flow graphs

In Section 4.3 the various solutions to the all-zero row matrix problem posed in [43] are modeled and manipulated. Section 4.4 is concerned with the impact of programming constructs on programming style and documentation. The chapter concludes with section 4.5.

4.2. Models of Control Flow Structures

Control structures used and/or discussed in [2,20,32..35,43] were translated into infoMaps and presented with graphical control flow graphs in Fig. 4.2.1. Control flow component of an infoMap is represented by a set of nodes {**Node**} interconnected with characters **s**, **d**, **l**, **a** and **e**. The characters "**s**", "**d**", "**l**", "**a**"and "**e**" mean **s**ource, **d**estination, **l**oop, **a**ssertion and **e**xception respectively.

By examining control flow components of the infoMaps in Fig. 4.2.1. it is noticeable that all control flow graphs are build from the four generic graphs given in Fig. 4.1.2. **If .. then .. else** constructs (see Fig. 4.2.1 (a)(b) and (c)) show simple tree-like structure with all leaves having a common exit node. These constructs are represented in the infoMaps by "**s**" and "**d**" pairs. **Loop-exit** - like structures (i.e. **while .. do .., repeat .. until .., for .. do ..**) are more complicated (see Fig. 4.2.1 (d)(e)(f)(g)(h)(i) and (j)). These structures are constructed in the infoMaps with "**s**" and "**d**" pairs, "**l**", and "**l**" and "**e**" pairs.

Pascal-type notation

```
if V then begin
        S1;
        S2
end;
```

Pancode notation

```
if V
   S1
   S2
```

EPN notation

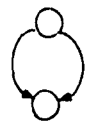
[V:S1;S2]

infoMap notation

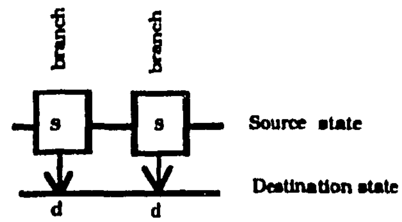
```
L  L  {Node}
s  s
d  d
G  G  {Precondition}
t  t    V
S  S  {Action Statement}
1       S1
2  .    S2
```

Control flow graphs

Conventional



Schematic



(a). if .. then ..

Pascal-type notation

```
if V1 then
        S1
else if V2 then begin
        S2
        S3
        end
    else
        S4;
```

Pancode notation

```
if V1
        S1
else if V2
        S2
        S3
     else
        S4
```

EPN notation

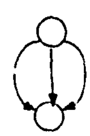
[V1:S1|V2:S2;S3|S4]

infoMap notation

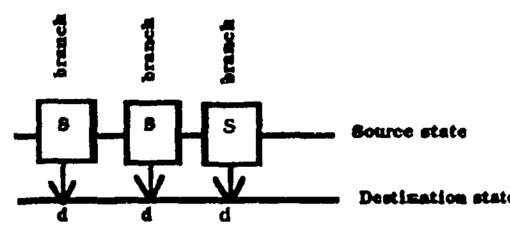
```
L  L  L  {Node}
s  s  s    .
d  d  d    .
G  G  G  {Precondition}
t  t  t     V1
.  t  t     V2
S  S  S  {Action Statement}
1  .        S1
   1        S2
.  2        S3
.  .  1     S4
```

Control flow graphs

Conventional



Schematic



(b). If .. then .. elseif ..then ..else

Fig. 4.2.1 Programming constructs

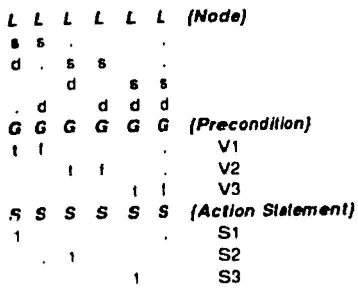
Pascal-type notation	**Pancode notation**	**EPN notation**

```
if V1 then begin
        S1;
        if V2 then begin
                S2;
                if V3 then begin
                        S3
                end
        end
end;
```

```
if V1
   S1
   also if V2
          S2
          also if  V3
                 S3
```

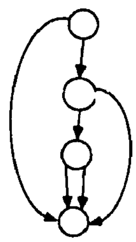
```
[V1:S1;V2:S2;V3:S3]
```

infoMap notation

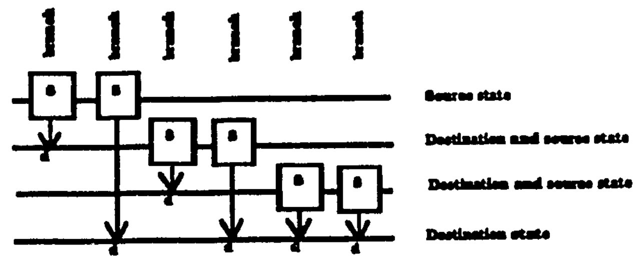


Control flow graphs

Conventional



Schematic



(c) if .. then .. if ..then ..if .. then ...

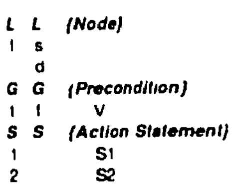
Pascal-type notation	**Pancode notation**	**EPN notation**

```
While V do begin
        S1;
        S2
end;
```

```
if V
   S1
   S2
repeat
```

```
{V:S1;S2|EXIT}
```

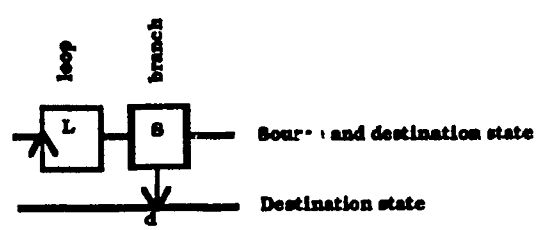
infoMap notation



Control flow graphs

Conventional

Schematic



(d). While .. do ..

Fig. 4.2.1 Programming constructs (cont.)

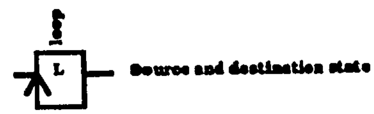
Pascal-type notation	Pancode notation	EPN notation

While true do begin
    S1;
    S2
end;
	do
	  S1
	  S2
	repeat
		{S1;S2}

Control flow graphs

infoMap notation	Conventional	Schematic

L *{Node}*
I
S *{Action Statement}*
1 S1
2 S2



(e). infinite loop

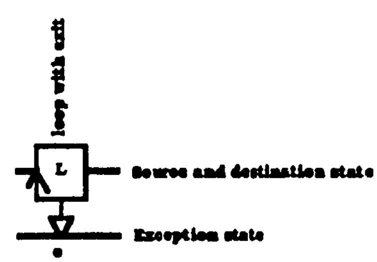
Pascal-type notation	Pancode notation	EPN notation

for k:=first to last by step do begin
    S1
    S2
end;
	for k:=first to last
	    S1
	    S2
	    next k:=k+step
		{[~V:EXIT]S1;S2;next k}
		    where,
		    V = k:=first to last

Control flow graphs

infoMap notation	Conventional	Schematic

L *{Node}*
I
e .
S *{Action Statement}*
1 S1
2 S2
3 k =k+step
G *{Postcondition}*
1 first=<k<=last



(f). For-loop

Fig. 4.2.1 Programming constructs (cont.)

Pascal-type notation

```
repeat
    S1;
    S2
until not V
```

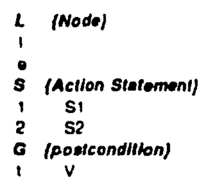
Pancode notation

```
do
  S1
  S2
also if V
  repeat
```

EPN notation

{S1;S2[~V:EXIT]}

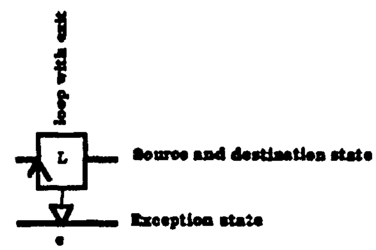
infoMap notation



Control flow graphs

Conventional

Schematic



(g). repeat .. until ..

Pascal-type notation

```
S1;
while V do begin
    S2;
    S1
end
```

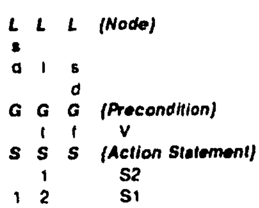
Pancode notation

```
do
  S1
also if V
  S2
  repeat
```

EPN notation

{S1[~V:EXIT]S2}

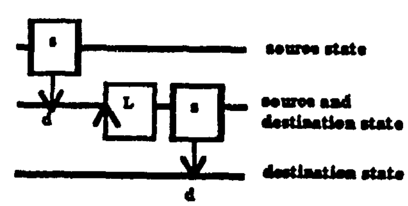
infoMap notation



Control flow graphs

Conventional

Schematic



(h). while .. do .. Loop

Fig. 4.2.1 Programming constructs (cont.)

Pascal-type notation	**Pancode notation**	**EPN notation**

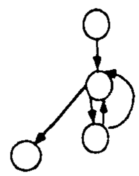
```
done:=false
while V1 and not done do begin
        S1;
        if V2 then
                S2
        else
                done:=true
end;
```

```
if V1
        S1
also if V2
        S2
        repeat
```

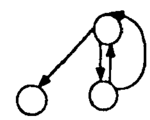
{[~V1:EXIT]S1[~V2:EXIT]S2]

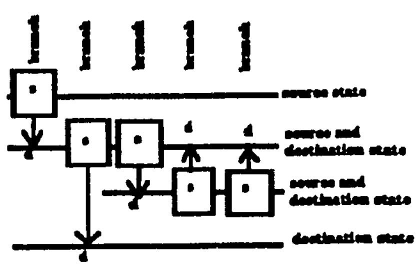
(1) **infoMap notation** "as is" infoMap	**Control flow graphs**	(2) **infoMap Notation** optimized infoMap	**Control flow graphs** optimized graph

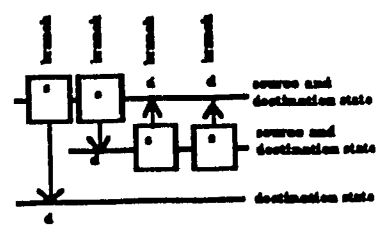
```
L  L  L  L  L   (Node)
s
d  s  s  d  d
.  .  d  s  s
   d                 .
G  G  G  G  G   (Precondition)
   1  1  .          V1 and not done
   .     1  1       V2
S  S  S  S  S   (Action Statement)
1              .    done =false
   .  1             S1
      .  1          S2
   .     .  1       done =true
```



```
L  L  L  L   (Node)
s  s  d  d
d     s  s      .
   d
G  G  G  G   (Precondition)
1  1  .         V1
      1  1      V2
S  S  S  S   (Action Statement)
1               S1
      1         S2
```







-- <u>(i). Loop .. exit ..exit</u> --

Pascal-type notation	**Pancode notation**	**EPN notation**

```
done:=false;
while V1 and not done do begin
        S1;
        S2;
        if not V2 then done:=true
end;
```

```
if V1
        S1
        S2
also if V2
        repeat
```

{[~V1:EXIT]S1;S2[~V2:EXIT}

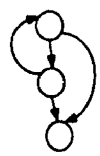
Fig. 4.2.1 Programming constructs (cont.)

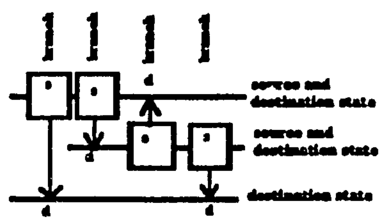
(1)

infoMap notation — "as is" infoMap

```
L L L L  {Node}
s s d      .
d   s s
  d   d
G G G G  {Precondition}
t f        V1
    t f    V2
S S S S  {Action Statement}
1          S1
2          S2
```

Control flow graphs



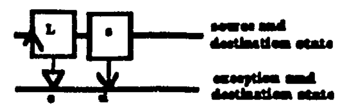


(2)

infoMap Notation — optimized infoMap

```
L L  {Node}
l s    .
e d    .
G G  {Precondition}
t f    V1
S S  {Action Statement}
1 .    S1
2 .    S2
G G  {Postcondition}
t .    V2
```

Control flow graphs — optimized graph



(j). Another loop

Fig. 4.2.1 Programming constructs (cont.)

There is a tradeoff between the number of nodes/states and the complexity of branches/transitions. A transition fork increases the transition complexity but reduces the number of states (compare Fig. 4.2.1 (j1) with Fig. 4.2.1 (j2)).

A transition with fork is named a **guaranteed function** . The guaranted function has a guard and a post-guard (i.e. an assertion or goal). The guaranted function is represented, at control flow level of infoMap, by "**l**" and "**a**" pair, or "**l**" and "**e**" pair, or "**s**" and "**a**" and "**e**" triple. The guaranteed function is an extension of the Guarded Command construct [18]. It is suggested that an often needed conditional exit from a loop should be represented by a guaranteed function nore elegant construct, syntactically and semantically .

4.3 Programming Style

A simple example used in [43] and discussions in [2] illustrate difficulty of writing an elementary Pascal program acceptable to a community of programmers. The same example is used in [35] to compare different programming constructs offered by Pancode and EPN. This example and solutions in [2,35,43] are used to illustrate the difficulty of manipulating and comparing programs build with conventional, string based languages and conventional programming constructs.

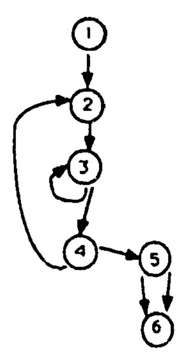
The problem, as stated in [18], is:

"Let X be a N x N matrix of integers. Write a program that will print the number of the first all-zero row of X, if any".

A solution from [18] and its control flow graph are given in Fig. 4.3.1 (a) and (b). The program was translated into infoMap (see Fig. 4.3.1(c)).

By processing rows and columns of the infoMaps, an optimized infoMap (Fig. 4.3.1 (d)), optimized control flow (Fig. 4.3.1 (e)) and optimized source code (Fig. 4.3.1 (f)) were produced. The solutions in Pancode and EPN taken from [35] and equivalent infoMaps are given in Appendix II (see Fig. 4.3.2 (a) and (b), and Fig. 4.3.3 (a)(b)(c)(d) and (e)). The infoMap based solutions could be further processed to obtain satisfactory or optimized infoMap.

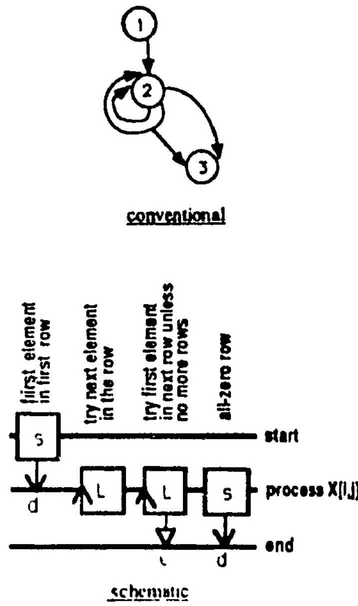
```
1:   i := 1;
2:  repeat
      j := 1;
3:   while(j <= n
      and (x[i, j] = 0) do
          j := j + 1;
     i := i + 1;
4: until (i > n) or (j > n);
5: if j > n
    then writeln
     ('The .. row is', i - 1 );
6:
```



L	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*(Node)*
s				.		.	.	1
d	s			d		.	.	2.
	d	l	s	.			.	3·
			d	s	s	.	.	4·
		.		.	d	s	s	5·.
				.		d	d	6:.
G	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*(preCondition)*
		t	f	.		.	.	j<=n and x[i,j]=0
		.	.	f	t		.	i>n or j>n
						t	f	j>n
S	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*(Action Statement)*
1								i =1
	1				.			j =1
		1		.	.			j =j+1,
			1	.	.	1		i =i+1;
				.		.	.	Print ('The . row is', i-1)

a) Pascal-type code b) control flow graph c) normalized infoMap

```
O  O  O  O  {Branch}
o              first  element  in  first  row
   o        .  try  next  element  in  the  row
      o     .  try  next  row  unless  no  more
      .  o     all-zero  row
L  L  L  L  {Node}
s     .  .     start
d  l  l  s     process  X[i,j]
.     e  d     end
G  G  G  G  {preCondition}
.  t  .  f     j<=n
.  t  f        x[i,j]=0
S  S  S  S  {Action  Statement}
1     .  .     i:=1
1     1        j =1
   1     .     j:=j+1
      1        i:=i+1
         1     Print  ('The  . row  is  ',  i-1)
G  G  G  G  {postCondition}
      t        i>n
```




```
1:  i:=1; j:=1;
2:  if j<=n and x[i,j]=0 then
        j:=j+1;          goto 2;
    if x[i,j]<>0 then
        j:=1; i:=i+1;
        if i>n then  goto 3 else
                         goto 2;
    if j>n then
        Print ('The .. row is', i-1);
                         goto 3;
3:
```

optimized

d) optimized infoMap e) control graph f) generated source code

Fig. 4.3.1 All zero row program

4.4. Constructs, Styles and Documentation

A summary of different solutions is given in Fig. 4.4.1.

{Version}	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*
Optimized	v	.	.	.	.	.	.	.	.	.	.	.		
Pancode	.	v	.	.	.	.	.		.	.	.	.	.	.
EPN-long	.	.	v	.	.	.	.	.	.	.	.	.	.	.
EPN-short	.	.	.	v	.	.	.	.	.	.	.	.	.	.
[1,14]	.	.	.	.	v	v	v	v	v	v	v	v	v	v
{Attribute}	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*
version # in [1,14]	.	.	.	.	1	2	3	4	5	6	7	8	9	10
branch	2	5	5	5	3	2	7	4	2	2	3	3	4	2
branch with fork	0	0	1	0	0	0	0	0	0	0	0	0	0	0
loop	1	1	0	1	2	0	1	0	0	0	2	2	2	0
loop with exit	1	0	0	0	0	2	0	2	2	2	0	0	0	2
state	3	4	5	4	3	4	8	4	3	4	3	3	4	4
transition	4	6	6	6	5	4	8	6	4	4	5	5	6	4
preCondition	2	3	3	3	3	1	4	3	2	2	3	3	3	2
action	6	5	5	5	5	5	5	5	5	5	5	5	5	5
postCondition	1	0	1	0	0	2	0	2	1	2	0	0	2	2

Fig. 4.4.1 Program attributes for the all-zero row program

The Pancode solution for the All-Zero Row problem represented as an infoMap has 4 nodes/states, 6 branches/transitions, 3 preconditions and 5 actions. The original EPN solution has 5 nodes/states, 6 branches/transitions, 3 preconditions, 5 actions and 1 postcondition. The Shorter EPN solution has 4 nodes/states, 6 branches/transitions, 3 preconditions and 5 actions (see Fig. 4.3.2 (a) and (b), and Fig. 4.3.3 (a)(b)(c)(d) and (e) in Appendix 4).

A Pascal solution taken from [43] and represented as an infoMap (compare Fig 4.3.1 (c)) has 6 nodes/states, 8 branches/transitions, 4 preconditions and 5 actions. Optimized infoMap solution (see Fig. 4.3.1 (c)) has 3 documented nodes/states, 4 documented branches/transitions, 2 preconditions, 5 actions and 1 postcondition.

The optimized solution was produced by merging states, transitions and by eliminating redundant preconditions and actions. The states and transitions in the optimized solutions are easy to document which might suggest that semantically meaningful states and transitions are produced by the optimization process.

First redundant actions and conditions are eliminated, and states and transitions merged. In the final step, possible tradeoff between complexity of transitions and number of states and transitions are considered. A transition fork increases the branch complexity but reduces the number of states and transitions (compare Fig. 4.2.1 (i1) with Fig. 4.2.1 (i2), Fig. 4.2.1 (j1) with Fig. 4.2.1 (j2)) and Fig. 4.3.1 (b) with Fig. 4.3.1 (d)). This optimization process is driven by a goal of creating an elegant solution from the semantically meaningful components. Therefore, it has to be driven by a human.

The analysis does not support superiority of EPN over Pancode claimed in [35]. The analysis of different Pascal solutions proposed in [2,35,43] shows that the programs are not built from semantically meaningful components and contain control flow graphs with redundant states and transitions. Conventional programming languages (f.ex. FORTRAN, Pascal, C, Pancode, EPN, etc) are not appropriate tools for the specification and processing of program structures.

4.5. Conclusion

The usefulness of infoMap as a tool for analysis of programs written in programming languages with different constructs has been demonstrated. Its modeling and processing power is evident in the representations of various constructs and styles. Most of our arguments related to programming constructs are also valid for system specification and design constructs. The system development processes transform initial objects into intermediate and final system products. The high degree of processability of the created objects is needed to assure high productivity and system products with high quality [50]. The infoMap notation is based on the fundamental notions of relations, graphs and functions. The infoMaps notational technology exploits the popular, simple and standardized spreadsheet environments. The infoMaps objects are processable.

The analysis and development of programs in this chapter is limited to control flow issues. Definition of data flow and data structures by source code might be satisfactory for simple programs. For larger programs it is necessary to model, analyze and optimize data flow graph and data structures. Tradeoffs between control flow, data flow and data model are necessary at the structural and conceptual level. To perform this task more sophisticated infoSchemas are needed (see Fig. 3.2.1 of chapter 3). For object oriented programs other infoSchemas are used [30].

Programming activities and objects are influenced by other phases of system life cycle. A notational technology should support system developers and managers during the whole system life cycle by providing System Information Space [50]. The infoMaps technology is a step in this direction [25,28..31].

Chapter 5

Application of TAPi to Algorithms

In this chapter the transformation of three sets of algorithms are shown using the TAPi methodology. This transformation is presented in three sections as follows:

5.1 String Search algorithms - this includes five algorithms and four procedures. The transformation of one algorithm is presented in this section; the others in Appendix III.

5.2 Degree-Constrained Minimum Spanning Tree algorithms - this includes three algorithms and one procedure. One transformed algorithm is shown in this section; the others in Appendix IV.

5.3 Skeletonization of Binary patterns algorithm - this includes two procedures and two functions. The transformation of one procedure is presented in this section; the other procedure and two functions are shown in Appendix V.

5.1 String Search algorithms

5.1.1. Introduction

The algorithms described in [48,49] are solutions to the string searching problem. The problem is:

"Given an input text of length n and a pattern of length m representing the pattern to be sought, find the (non-) occurrence of the pattern in the text".

The problem statement may be represented as an infoMap shown in Fig. 5.1.1.1.

A C {Problem}
v searching for the occurrence(s) of a pattern within a text
S 3 {Sentence}
1 Given an input text of length n and a pattern of
2 length m representing the pattern to be sought,
3 find the (non-) occurrence of the pattern in the text.

Fig. 5.1.1.1 String Search Problem Statement (Narrative)

The purpose here is not to describe these algorithms in detail but to attempt to transform them using TAPi. In so doing, a user would not only discover if inheritance does exist but also, if components are reusable. The process involves the building of the various models mentioned in chapter 3. Therefore, the structures represented in Figures 3.3.1, 3.4.1, 3.5.1 and 3.6.1 would be used in this modeling process.

The TAPi methodology emphasizes the need to consider not only the control flow but also the hierarchical model, the data model and data flow model-building aspects. The last three modeling aspects are demonstrated in section 5.1.2.

The control flow aspect of string search algorithms and their transformations into infoMaps are presented and discussed in section 5.1.3.

The main deliverable from the process is an optimized and well-documented search algorithm. The intermediate steps should not be neglected since they are crucial to the achievement of this deliverable. Section 5.1.4 summarizes and presents statistics on the various components of the algorithms, both of section 5.1 and appendix III.

5.1.2. Hierarchy, Data Model and data flow

The hierarchical model for string search algorithms are shown in Fig. 5.1.2.1. The Straight Forward algorithm does not call any procedure; the others do. This is indicated by "p" in column 1. The KMP algorithm needs one procedure - computing table for KMP pattern matching; whereas Substring Search needs two - computing table TD1[] and computing table TD2[].

	H	H	H	H	H	5	*(Algorithm)*
	p	.	.	.	.	.	Straight Forward (SF)
	.	h	.	.	.	.	Knuth-Morris-Pratt (KMP)
	.	.	h	.	.	.	Boyer-Moore (BM)
	.	.	.	h	.	.	A Substring Search (SS)
	.	.	.	.	h	.	Quick Search (QS)
(Procedural Statement)	.	.	.	.	.	4	*(Procedure)*
	.	1	.	.	.		computing table for KMP pattern matching
	.	.	1	.	.		computing DELTA1 and DELTA2
	.	.	.	1	1		computing table TD1[]
	.	.	.	2	.		computing table TD2[]

Fig. 5.1.2.1 Hierarchical view of string search algorithms

The next step in TAPi is to attempt to recover the data model. The transformed data model for the algorithms is presented in Fig. 5.1.2.2. It is apparent that many attributes/data-objects are common amongst the algorithms; thus indicating that inheritance does exist.

	o	*o*	*o*	*o*	*o*	.	.	5	***{Algorithm}***
	o	.	.	.	.	.	.	.	StraightForward
	.	o	.	.	.	.	.	.	Knuth-Morris-Pratt
	.	.	o	.	.	.	.	.	Boyer-Moore
	.	.	.	o	.	.	.	.	A Substring Search
	.	.	.	.	o	.	.	.	QuickSearch
{Declarative Statement}	.	.	.	.	.	*o*	*o*	2	***{Type}***
character	.	.	.	.	.	o	.	.	character
integer	.	.	.	.	.	.	o	.	integer
{Declarative Statement}	***M***	***M***	***M***	***M***	***M***	***M***	***M***	24	***{Data-object/Attribute}***
i	a	.	.	.	.	.	v		guess ptr. for pat within the text
i[]	.	.	.	a	.	.	v		an ordered pattern
j	.	a	a	a	.	.	v		scan pointer
n	a	a	a	a	a	.	v		length of text string
m	a	a	a	a	a	.	v		length of pattern substring
PAT[]	a	a	a	a	a	v	.		pattern substring
TXT[]	a	a	a	a	a	v	.		text string
k	a	.	.	.	.	.	v		subscript indicating current pos in pattern
k	.	a	a	.	a	.	v		subscript indicating current pos in text
j	a	.	.	.	.	.	v		subscript indicating current pos in text
qsearch:=k	a	.	.	a	a	.	v		"pattern found at"
qsearch:= -1	a	.	.	a	a	.	v		"pattern not found"
FAIL()	.	a	.	.	.	.	v		failure table for KMP algorithm
DELTA1()	.	.	a	.	.	.	v		array computed from the function DELTA1
DELTA2()	.	.	a	.	.	.	v		array computed from the function DELTA2
gotmatch	.	.	.	a	a	v	.		match/no-match flag
lshift	.	.	.	a	.	.	v		leftward shift
matchshift	.	.	.	a	.	.	v		returns value of the next leftward shift
gotshift	.	.	.	a	.	v	.		got the correct shift
TD1[]	.	.	.	a	a	.	v		array for obtaining required shift
TD2[]	.	.	.	a	.	.	v		array for obtaining required maximal shift
delta1	.	.	.	a	.	.	v		amount of shift p to the right
delta2	.	.	.	a	.	.	v		pos in pat where mismatch 1st occur

Fig. 5.1.2.2 Data Model for string search algorithms

In Fig. 5.1.2.3 the data flow within a Straight Forward search algorithm (see Fig. 3.2.3) is shown. This model shows the data-objects as they flow in and/ or out of conditions and actions. The "**i**", "**o**" and "**b**" mean input, output and both input and output respectively. For example, in the statement "i:= i+1", i is both input and output, hence the "**b**".

	A	A	A	A	A	A	A	A	A	1	*(Algorithm)*
	v	v	v	v	v	v	v	v	v		Straight Forward
(Declarative Statement)	O	O	O	O	O	O	O	O	O	9	*(Attribute/data-object)*
i	o	.	.	.	.	.	.	.	.		guess ptr. for pat within the text
n	.	o	.	.	.	.	.	.	.		length of text string
m	.	.	o	.	.	.	.	.	.		length of pattern substring
PAT()	.	.	.	o	.	.	.	.	.		pattern substring
TXT()	.	.	.	.	o	.	.	.	.		text string
k	.	.	.	.	.	o	.	.	.		subscript indicating current pos in pat.
j	.	.	.	.	.	.	o	.	.		subscript indicating current pos in text
"pattern found at"	.	.	.	.	.	.	.	o	.		success message
"pattern not found"	.	.	.	.	.	.	.	.	o		failure message
(Conditional Statement)	F	F	F	F	F	F	F	F	F	3	*(preCondition)*
i<n-m+1	i	i	i	.	.	.	.	.	.		location within text
PAT(k) = TXT(j)	.	.	.	i	i	i	i	.	.		pat elem equals text elem
k = m	.	.	i	.	.	i	.	.	.		all pat chars compared
(Imperative Statement)	F	F	F	F	F	F	F	F	F	8	*(Action)*
i := 0;	o	.	.	.	.	.	.	.	.		set guess ptr. to zero
i := i+1	b	.	.	.	.	.	.	.	.		increment guess ptr. by 1
j := i;	i	.	.	.	.	.	o	.	.		set text ptr. to guess ptr. value
k := 1;	.	.	.	.	.	o	.	.	.		set pat ptr. to 1
return("pattern found at", i)	i	.	.	.	.	.	.	o	.		pattern found at guess ptr. value
j := j+1;	.	.	.	.	.	.	b	.	.		increment text ptr. by 1
k := k+1;	.	.	.	.	.	b	.	.	.		increment pat ptr. by 1
return("pattern not found")	.	.	.	.	.	.	.	.	o		pattern not found
(Conditional Statement)	F	F	F	F	F	F	F	F	F	.	*(postCondition)*

Fig. 5.1.2.3 Data flow within a Straight Forward search algorithm

5.1.3 Control flow

The main goal in transforming the "control" aspect of the algorithms is not only to produce optimized and documented versions but also to trace inheritance amongst them. By adding the cardinality of each set of the infoSchema of Fig. 3.6.1 , the resulting infoSchema is given in Fig. 5.1.3.1. This infoSchema would be used to normalize, optimize, document and trace inheritance amongst the algorithms.

	A	A	4	{Algorithm}
	v	v		Straight Forward
	v	v		Knuth-Morris-Pratt
	v	v		Boyer-Moore
	v	v		Substring Search
	v	v		Quick search
	O	.	x	{Path}
	S	O	6	{Transition}
	.	L	4	{State}
{Conditional Statement}	.	G	3	{preCondition}
{Procedural Statement}	.	S	4	{Procedure}
{Imperative Statement}	.	S	x	{Action}
{Conditional Statement}	.	G	.	{postCondition}

Fig. 5.1.3.1 An infoSchema of the control flow of string search algorithms

The algorithms as they go through the different stages of transformation are shown below in Fig. 5.1.3.2 and also in Fig. 5.1.3.3 to Fig. 5.1.3.10 of Appendix III.

The optimized representations are produced by merging of states, transitions and by eliminating redundant preconditions and actions. First, redundant actions and conditions are eliminated and states and transitions merged. In the final step, possible tradeoff between complexity of transitions and the number of states and transitions are considered. The optimization process is driven by a goal of creating an elegant solution from sematically meaningful components.

A Straight Forward search algorithm is presented below to demonstrate the methodology's control flow modeling aspect (Fig. 5.1.3.2). The steps of this transformation process presented and discussed in chapter 3 are followed. The control flows for the rest of the algorithms are shown in Fig.5.1.3.3 to 5.1.3.10 of Appendix III.

```
1.    i := 0;
2.    while(i<n-m+1) do
3.          i := i+1
4.          j := i; k := 1;
5.          while(PAT(k) = TXT(j)) do
6.                if(k = m) then
7.                      return("pattern found at", i)
                  else do
8.                      j := j+1;
9.                      k := k+1;
                  end;
            end;
      end;
10.   return("pattern not found");
```

a) source code: original

```
1 :   i := 0;
2 :   while(i<n-m+1) do
            i := i+1
            j := i; k := 1;
3 :         while(PAT(k) = TXT(j)) do
4 :               if(k = m) then
                        return("pattern found at", i)
                  else do
                        j := j+1;
                        k := k+1;
                  end;
            end;
      end;
5 :   return("pattern not found");
```

b) edited source code: line nos removed, states 1..5 identified

	O	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	7	*(Transition)*
	.	o	.	.	.	.	.	.		
	.	.	o	.	.	.	.	.		
	.	.	.	o	.	.	.	.		
	.	.	.	.	o	.	.	.		
	.	.	.	.	.	o	.	.		
	.	.	.	.	.	.	o	.		
	.	.	.	.	.	.	.	o		
	.	L	L	L	L	L	L	L	5	*(State)*
	.	s	.	.	.	.	.	.		1:.
	.	d	s	s	.	d	.	.		2:.
	.	.	d	.	s	s	.	d		3:.
	.	.	.	.	d	.	s	s		4:.
	.	.	.	d	.	.	d	.		5:.
(Conditional Statement)	.	G	G	G	G	G	G	G	3	*(preCondition)*
i<n-m+1	.	.	t	f	.	.	.	.		location within text
PAT(k) = TXT(j)	.	.	.	.	t	f	.	.		pat elem equals text elem
k=m	.	.	.	.	.	.	t	f		all pat chars compared
(Imperative Statement)	.	S	S	S	S	S	S	S	8	*(Action)*
i := 0;	.	1	.	.	.	.	.	.		set guess ptr. to zero
i := i+1	.	.	1	.	.	.	.	.		increment guess ptr. by 1
j := i;	.	.	2	.	.	.	.	.		set text ptr. to guess ptr. value
k := 1;	.	.	3	.	.	.	.	.		set pat ptr. to 1
return("pattern not found");	.	.	.	1	.	.	.	.		pattern not found
return("pattern found at", i)	.	.	.	.	.	.	1	.		pattern found at guess ptr. value
j := j+1;	.	.	.	.	.	.	.	1		increment text ptr. by 1
k := k+1;	.	.	.	.	.	.	.	2		increment pat ptr. by 1
(Conditional Statement)	.	G	G	G	G	G	G	G	.	*(postCondition)*

c) Normalized infoMap: shaded areas to be filled in

	O	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	7	*(Transition)*
	.	o	.	.	.	.	.	.		initialize guess ptr. for pat within the text
	.	.	o	.	.	.	.	.		location within the text
	.	.	.	o	.	.	.	.		location outside the text
	.	.	.	.	o	.	.	.		pat elem matches with text elem
	.	.	.	.	.	o	.	.		pat elem does not match with text elem
	.	.	.	.	.	.	o	.		all pat chars matched
	.	.	.	.	.	.	.	o		calculate new pat and text ptrs.
	.	L	L	L	L	L	L	L	5	*(State)*
	.	s	.	.	.	.	.	.		1:.Start
	.	d	s	s	.	d	.	.		2:.evaluating remaining text
	.	.	d	.	s	s	.	d		3:.matching pat elem with text elem
	.	.	.	.	d	.	s	s		4:.testing for pattern occurrence
	.	.	.	d	.	.	d	.		5:.exit
(Conditional Statement)	.	G	G	G	G	G	G	G	3	*(preCondition)*
i<n-m+1	.	.	t	f	.	.	.	.		location within text
PAT(k) = TXT(j)	.	.	.	.	t	f	.	.		pat elem equals text elem
k=m	.	.	.	.	.	.	t	f		all pat chars compared
(Imperative Statement)	.	S	S	S	S	S	S	S	8	*(Action)*
i := 0,	.	1	.	.	.	.	.	.		set guess ptr. to zero
i := i+1	.	.	1	.	.	.	.	.		increment guess ptr. by 1
j := i;	.	.	2	.	.	.	.	.		set text ptr. to guess ptr. value
k := 1;	.	.	3	.	.	.	.	.		set pat ptr. to 1
return("pattern not found");	.	.	.	1	.	.	.	.		pattern not found
return("pattern found at", i)	.	.	.	.	.	.	1	.		pattern found at guess ptr. value
j := j+1;	.	.	.	.	.	.	.	1		increment text ptr. by 1
k := k+1;	.	.	.	.	.	.	.	2		increment pat ptr. by 1
(Conditional Statement)	.	G	G	G	G	G	G	G	.	*(postCondition)*

d) Normalized infoMap: states and transitions added (shaded areas)

	O	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	7	*(Transition)*
	.	o	.	.	.	.	.	.		initialize guess ptr. for pat within the text
	.	.	o	.	.	.	.	.		location within the text
	.	.	.	o	.	.	.	.		location outside the text
	.	.	.	.	o	.	.	.		pat elem matches with text elem
	.	.	.	.	.	o	.	.		pat elem does not match with text elem
	.	.	.	.	.	.	o	.		all pat chars matched
	.	.	.	.	.	.	.	o		calculate new pat and text ptrs.
	.	*L*	*L*	*L*	*L*	*L*	*L*	*L*	5	*(State)*
	.	s	.	.	.	.	.	.		1:.Start
	.	d	s	s	.	d	.	.		2:.evaluating remaining text
	.	.	d	.	s	s	.	d		3:.matching pat elem with text elem
	.	.	.	.	d	.	s	s		4:.testing for pattern occurrence
	.	.	.	d	.	.	d	.		5:.exit
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	3	*(preCondition)*
i<n-m+1	.	.	t	f	.	.	.	.		location within text
PAT(k) = TXT(j)	.	.	.	.	t	f	.	.		pat elem equals text elem
k=m	.	.	.	.	.	.	t	f		all pat chars compared
(Imperative Statement)	.	*S*	*S*	*S*	*S*	*S*	*S*	*S*	8	*(Action)*
i := 0;	.	1	.	.	.	.	.	.		set guess ptr. to zero
i := i+1	.	.	1	.	.	.	.	.		increment guess ptr. by 1
j := i;	.	.	2	.	.	.	.	.		set text ptr. to guess ptr. value
k := 1;	.	.	3	.	.	.	.	.		set pat ptr. to 1
return("pattern not found");	.	.	.	1	.	.	.	.		pattern not found
return("pattern found at", i)	.	.	.	.	.	.	1	.		pattern found at guess ptr. value
j := j+1;	.	.	.	.	.	.	.	1		increment text ptr. by 1
k := k+1;	.	.	.	.	.	.	.	2		increment pat ptr. by 1
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	.	*(postCondition)*

e) Normalized infoMap: state diagram (unshaded area)

	O	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	7	*(Transition)*
	.	o	.	.	.	.	.	.		initialize guess ptr. for pat within the text
	.	.	o	.	.	.	.	.		location within the text
	.	.	.	o	.	.	.	.		location outside the text
	.	.	.	.	o	.	.	.		pat elem matches with text elem
	.	.	.	.	.	o	.	.		pat elem does not match with text elem
	.	.	.	.	.	.	o	.		all pat chars matched
	.	.	.	.	.	.	.	o		calculate new pat and text ptrs.
	.	*L*	*L*	*L*	*L*	*L*	*L*	*L*	5	*(State)*
	.	s	.	.	.	.	.	.		1:.Start
	.	d	s	s	.	d	.	.		2:.evaluating remaining text
	.	.	d	.	s	s	.	d		3:.matching pat elem with text elem
	.	.	.	.	d	.	s	s		4:.testing for pattern occurrence
	.	.	.	d	.	.	d	.		5:.exit
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	3	*(preCondition)*
i<n-m+1	.	.	t	f	.	.	.	.		location within text
PAT(k) = TXT(j)	.	.	.	.	t	f	.	.		pat elem equals text elem
k=m	.	.	.	.	.	.	t	f		all pat chars compared
(Imperative Statement)	.	*S*	*S*	*S*	*S*	*S*	*S*	*S*	8	*(Action)*
i := 0;	.	1	.	.	.	.	.	.		set guess ptr. to zero
i := i+1	.	.	1	.	.	.	.	.		increment guess ptr. by 1
j := i;	.	.	2	.	.	.	.	.		set text ptr. to guess ptr. value
k := 1;	.	.	3	.	.	.	.	.		set pat ptr. to 1
return("pattern not found");	.	.	.	1	.	.	.	.		pattern not found
return("pattern found at", i)	.	.	.	.	.	.	1	.		pattern found at guess ptr. value
j := j+1;	.	.	.	.	.	.	.	1		increment text ptr. by 1
k := k+1;	.	.	.	.	.	.	.	2		increment pat ptr. by 1
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	.	*(postCondition)*

f) Normalized infoMap to be optimized: shaded area

	O	.	.	.	.	.	.	x	*{Path}*
	S	O	O	O	O	O	O	6	*{Transition}*
	.	o	.	.	.	.	.		initialize guess ptr. for pat within the text
	.	.	o	.	.	.	.		location within the text
	.	.	.	o	.	.	.		location outside the text
	.	.	.	.	o	.	.		all pat chars matched
	.	.	.	.	.	o	.		match at current text location
	.	.	.	.	.	.	o		pat elem does not match with text elem
	.	L	L	L	L	L	L	4	*{State}*
	.	s	.	.	.	.	.		1:.Start
	.	d	s	s	.	.	d		2:.evaluating remaining text
	.	.	d	.	s	l	s		3:.matching pat elem with text elem
	.	.	.	d	d	.	.		4:.exit
{Conditional Statement}	.	G	G	G	G	G	G	3	*{preCondition}*
i<n-m+1	.	.	t	f	.	.	.		location within text
PAT(k) = TXT(j)	.	.	.	.	t	t	f		pat elem equals text elem
k=m	.	.	.	.	t	f	.		all pat chars compared
{Imperative Statement}	.	S	S	S	S	S	S	8	*{Action}*
i := 0;	.	1	.	.	.	.	.		set guess ptr. to zero
i := i+1	.	.	1	.	.	.	.		increment guess ptr. by 1
j := i;	.	.	2	.	.	.	.		set text ptr. to guess ptr. value
k := 1;	.	.	3	.	.	.	.		set pat ptr. to 1
return("pattern not found");	.	.	.	1	.	.	.		pattern not found
return("pattern found at", i)	.	.	.	.	1	.	.		pattern found at guess ptr. value
j := j+1;	.	.	.	.	.	1	.		increment text ptr. by 1
k := k+1;	.	.	.	.	.	2	.		increment pat ptr. by 1
{Conditional Statement}	.	G	G	G	G	G	G	.	*{postCondition}*

g) Optimized and documented infoMap

1:. Start
initialize guess ptr. for pat within the text, CONTINUE AT 2

2:. evaluating remaining text
location within the text, CONTINUE AT 3
location outside the text, CONTINUE AT 4

3:. matching pat elem with text elem
all pat chars matched, CONTINUE AT 4
match at current text location, CONTINUE AT 3
pat elem does not match with text elem, CONTINUE AT 2

4:. exit

h) Optimized State diagram as Structured English (generated)

1:. Start
initialize guess ptr. for pat within the text, CONTINUE AT 2
i:=0; goto 2

2:. evaluating remaining text
location within the text, CONTINUE AT 3
i < n-m+1 -> i:= i+1; j:= i; k:= 1; goto 3

location outside the text, CONTINUE AT 4
i>= n-m+1 -> return("pattern not found"); goto 4

3:. matching pat elem with text elem
all pat chars matched, CONTINUE AT 4
PAT(k) = TXT(j) and k=m -> return("pattern found at", i); goto 4

match at current text location, CONTINUE AT 3
PAT(k) = TXT(j) and k<>m -> j:= j+1; k:= k+1; goto 3

pat elem does not match with text elem, CONTINUE AT 2
PAT(k) <> TXT(j) -> goto 2

4:. exit

i) source code (generated)

Fig. 5.1.3.2 Control flow for a Straight Forward search algorithm

A comparison of the control flows for string search optimized algorithms is given in Fig.5.1.3.11. Fig. 5.1.3.11 shows that algorithms share common components such as transitions, states and conditions. These algorithms have identical control flow and guards. The algorithms differ in procedures and actions; however, they do share some of these.

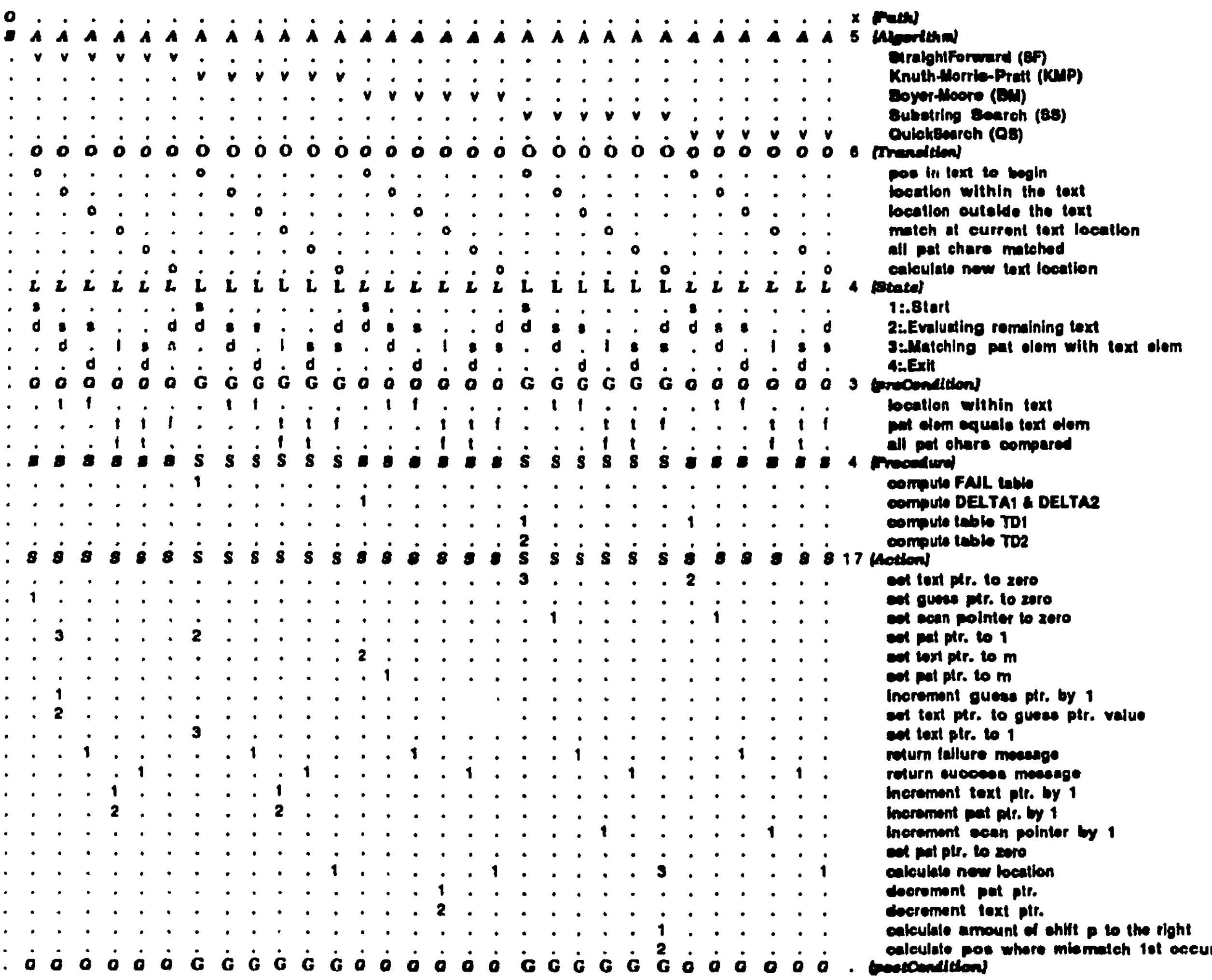


Fig. 5.1.3.11 Control flows (code column hidden) for string search optimized algorithms

5.1.4 Summary and Conclusions

A comparison of the data flows for string search algorithms is given in Fig. 5.1.4.1. Many common preconditions and actions are noticeable. Even though it may be claimed that the algorithms were built from scratch, the results of this transformation prove otherwise.

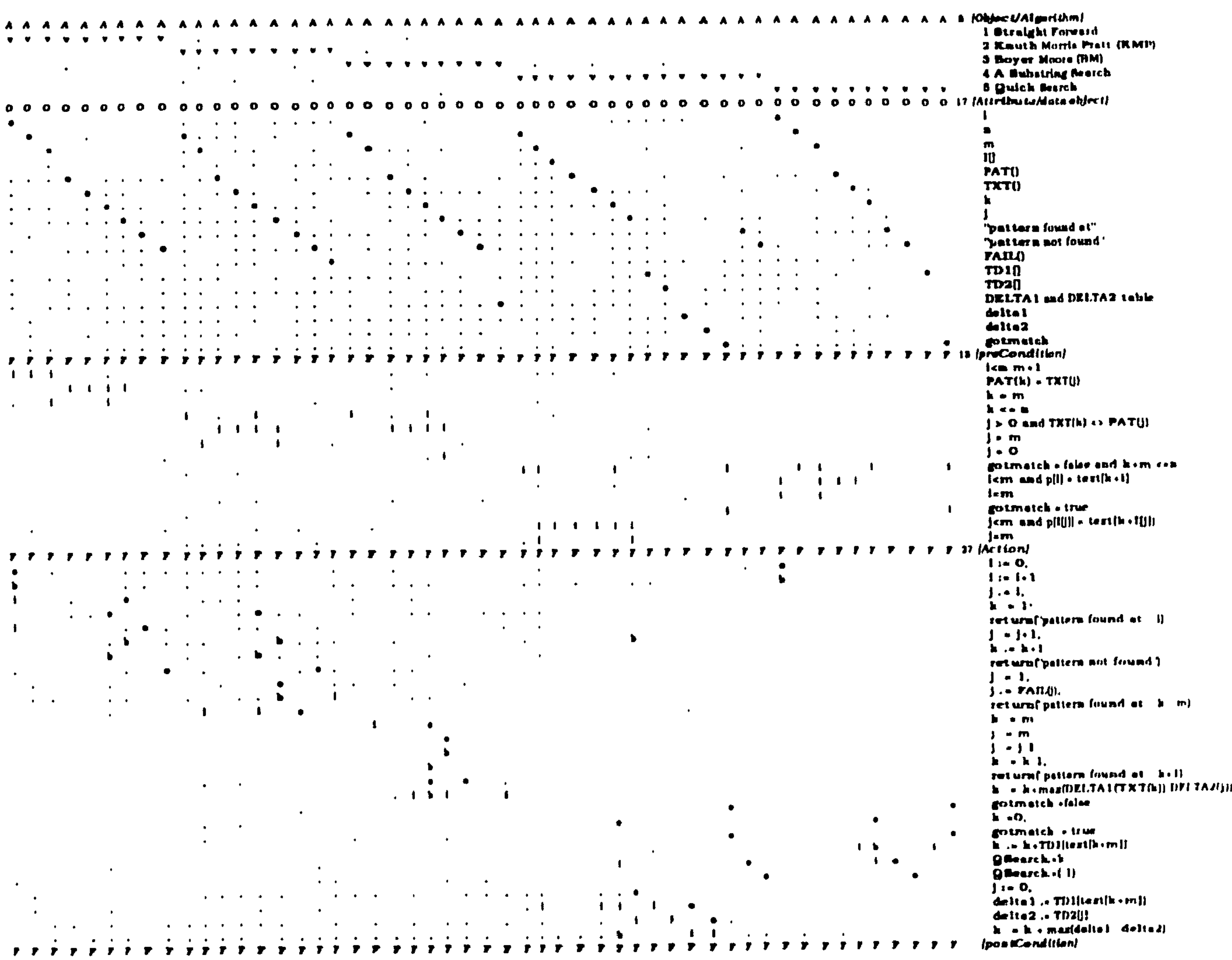


Fig. 5.1.4.1 Data flows for string search algorithms

A comparison of the control flows of string search algorithms is given in Fig. 5.1.4.2. Each algorithm is cptimized and in some cases very much significant. For example the number of transitions for the Knuth-Moore-Pratt process has been reduced from a total of 14 to 11. The complexity of the procedures is also shown in this figure. For instance, the procedure - BM DELTA1 & DELTA2 - has 16 transitions reduced to 13. Procedures and algorithms with cardinality greater than nine is considered not simple.

(View)	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Straight Forward process	V	V		.	.	.	.	.	.	.			.	.		.	.	.	.	.
Knuth-Morris-Pratt process	.		V	V	V	V			.	.			.	.		.	.		.	.
Boyer-Moore process			.	.			V	V	V	V		.	.	.	.	.	.		.	.
Quick Search process		.		.		.			.	.	V	V	V	V			.	.	.	.
Search process	.	.		.	.	.		.	.	.	.		.	.	V	V	V	V	V	V
(Procedure)	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
KMP Failure Table	.	.	V	.	V	.	.				.		.	.	.		.	.	.	.
BM DELTA1 & DALTA2	.	.	.	.	.	.	V	.	V		.	.	.	.	.	.	.	.	.	.
Computing table TD1[]		.	.	.	.	.			.	.	V		V	.	V	.	.	V	.	.
Computing table TD2[]	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	V	.	.	V	.
(Algorithm)	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Straight Forward	V	V	.	.	.		.		.	.	.		.		.		.		.	.
Knuth Morris-Pratt (KMP)	.		.	V		V	.	.	.	.		.	.	.		.	.		.	.
Boyer-Moore (BM)	.	.	.		.	.	.	V	.	V		.	.	.		.			.	.
Quick Search	.	.		.	.	.	.	.	.	.		V	.	V		.	.	.	.	.
Substring Search	.	.		.	.	.	.		.	.			.	.		.	V	.	.	V
(Version)	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Normalized	V	.	V	V	.		V	V	.		V	V	.	.	V	V	V	.	.	.
Optimized	.	V		.	V	V	.		V	V			V	V	.	.	.	V	V	V
(Attribute)	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V
branch	7	5	6	6	6	5	12	6	6	5	2	8	.	5	2	10	8	.	9	5
branch with fork						.		.	.	.			.	.			.		.	.
loop		1	1	1		1	4	1	6	1	2	1	.	1	2	1	1		1	1
loop with exit		.			.	.		.	1	.			2	.				2	.	.
transition	7	6	7	7	5	6	16	7	13	6	4	9	2	6	4	11	9	2	10	6
state	5	4	5	5	4	4	9	5	7	4	3	6	3	4	3	6	6	3	6	4
preCondition	3	3	4	4	3	3	8	4	7	3	2	5	.	3	2	6	5	.	6	3
action	8	8	8	7	8	7	14	7	14	7	2	8	2	6	2	9	10	2	9	8
postCondition		.			.	.	.		1	.			2	.			.	2	.	.

Legend: ☐ *Normalized* *Optimized*

Fig. 5.1.4.2 Attributes of String Search algorithms

Optimization of the string search algorithms reduces their complexity which is one of the principal goals of research in Knowledge Engineering [11,19].

Fig.5.1.4.3 shows that the algorithms were built from reusable components; thus illustrating that inheritance does exist. The infoMaps show that the "Transition" and "State" sets for the algorithms are identical which suggest that the environments are the same. This also suggests that the algorithms were not built from scratch but modifications were made to already existing ones. It should be noted that in the figure each procedure is treated as an attribute of the algorithms and not as comprising of a set of attributes as is the case in Fig. 5.1.4.2.

(Algorithm)	*A*	*A*	*A*	*A*	*A*	*A*
Straight Forward	v	.	.		.	v
Knuth-Morris-Pratt (KMP)	.	v	.	.	.	v
Boyer-Moore (BM)	.	.	v	.	.	v
Quick Search	.	.	.	v	.	v
Substring Search	.	.	.	.	v	v
(Version)	*A*	*A*	*A*	*A*	*A*	*A*
Optimized	v	v	v	v	v	.
Common	.	.	.		.	v
(Attribute)	*V*	*V*	*V*	*V*	*V*	*V*
branch	5	5	5	5	5	5
branch with fork	.	.	.	.	.	.
loop	1	1	1	1	1	1
loop with exit	.	.	.	.	.	.
transition	6	6	6	6	6	6
state	4	4	4	4	4	4
preCondition	3	3	3	3	3	3
procedure	.	1	1	1	2	.
action	8	7	7	6	8	2
postCondition	.	.	.	.	.	.

Legend: ☐ optimized common

Fig. 5.1.4.3 Common attributes of String Search algorithms

Based on the foregoing analysis, it is apparent that the key difference amongst the algorithms is that some matches the pattern in the reverse direction from the direction in which the pattern is shifted. Others matches the pattern in a left to right direction. In other words they differ in procedures and actions.

The methodology employed here demonstrates that existing components or codes are reusable, thus reducing development time and cost.

5.2 Degree-Constrained Minimum Spanning Tree algorithms

5.2.1 Introduction:

In this section the TAPi methodology is used once more to transform algorithms. The algorithms under consideration are those offering solutions to the Degree-Constrained Minimum Spanning Tree (DCMST) problem. The problem [39] can be stated as follows:

"Given a non-directional complete graph G(V,E), with cost(length, time) C_{ij} associated with the edge e_{ij} for every e_{ij} element of E; construct a minimum cost spanning tree that the degree d_i at a node i for every i element of V, is less than or equal to b_i".

The problem statement can be transformed and represented as an infoMap as shown in Fig. 5.2.1.1

A C {Problem}
v   of constructing a degree-constrained minimum cost spanning tree
S 6 {Sentence}
1   Given a non-directional complete graph G(V,E),
2   with cost(length, time) C_{ij} associated with the
3   edge e_{ij} for every e_{ij} element of E;
4   Construct a minimum cost spanning tree that the degree
5   d_i at a node i for every i element of V, is less than
6   or equal to b_i.

Fig. 5.2.1.1 DCMST Problem Statement (Narrative)

Unlike the modeling in Section 5.1, a data flow is given for each algorithm. The notion is not to repeat the process but with each transformation of an algorithm emphasize various aspects of the methodology. Three algorithms and one procedure offered as solutions to the problem are considered. The various views for the Degree-Constrained Minimum Spanning Tree (DCMST) algorithms are presented in sections 5.2.2 to 5.2.4 and Appendix IV. These algorithms are opportune for the data flow transformation aspect of the methodology.

5.2.2 Data model

A data model for the Degree-Constrained Minimum Spanning Tree (DCMST) algorithms is presented in Fig. 5.2.2.1. Apart from identifying the attributes/ data-objects and their type for each object, the means whereby each attribute can be documented properly is provided. Thus at the very beginning, meaning is given to the coded attributes. At times, it may be difficult to give meaning to the coded attributes/ data-objects, therefore, one may have to consult the author of the algorithm or other appropriate sources. Gaps in giving meaning at this stage would most likely results in gaps when giving meaning to the various set members of the data flow and control flow models.

	O	*O*	*O*	*O*	*O*	*O*	4	*{Object/Algorithm}*
	o	.	.		.	.		DCST
	.	o	.		.	.		Primal
	.	.	o					Dual
	.		.	o	.	.		Anneal
{Declarative Statement}	.	.	.	.	*O*	*O*	2	*{dataTYPE }*
integer	.	.	.	.	o	.		integer
real	.	.	.	.	.	o		real number
{Declarative Statement}	*M*	*M*	*M*	*M*	*M*	*M*	34	*{Attribute/data-object}*
V	a	a	a	.	.	.		the set of nodes
i	a	.	a	a	v	.		1, ..., n; nodes of V
ini[]	a	a	a	.	.	.		Function; a node corresponding to j
j	a	a	a	a	v	.		a node in V
U[]	a	.	a	.	.	.		an array of weights
W	a	a	a	a	.	.		2-dimensional matrix of weights
P	a	.	a	.	.	.		a subset of nodes
S	a	a	a	a	.	.		a subset of edges in a graph
e	.	a	a	a	.	.		an edge
k	a		a		v			a node not in P
b	a	a	a	.	v	.		max. edges connected to a node
g	a	.			.			a node of P
g'	a				.	.		a node not in P
cost of S	a	a		a		v		summation of weights in a tree
n	a	a		a	v			no. of nodes in the graph
∞	a							two nodes with no edges between them
E		a	a	a	.			a set of edges
T		a	a	a	.			a subset of the nodes in a graph
w	.	a						a node of T
v		a			v	.		a node of T
d[]		a	a	a	v			degree of node
x	.		a	a	v	.		a node of T_j
y		.	a	a	v			a node of T_i
D	.	.	a	.				a matrix of nodes from T_i and T_j
DCST		a	a	a	.			Degree-Constrained Spanning Tree
f[]	.	.	a	.	.			final matrix
r			.	a	.	v		a random number between 0 nad 1
t	.		.	a	.	.		a temperature
f	.	.	.	a	.	.		a reducer
ß	.			a	.	v		beta, a constant use in temp calculation
Δ	.	.		a	.	v		a difference between two weights
Q1	.	.	.	a	.	.		queue with edges already in S
Q2	.		.	a	.	.		queue with edge not yet in S

Fig. 5.2.2.1 Data model for DCMST algorithms

5.2.3 Data flow:

The algorithm to be transformed is given in Fig. 5.2.3.1.

Inputs: $G = (V, E)$, $V = \{1, 2, \ldots, n\}$, $b > 0$.
$\forall i, j \in V$, w_{ij} is the weight between i and j.

Definitions: P : set of nodes already in DCST. $P \subseteq V$.
S : set of edges already in DCST. $S \subseteq E$.
$\forall i \in V$, degree$[i]$ is the current degree of i in DCST.
$\forall i \in V$, assume $w_{k'i} = \min_{k \in P} w_{ki}$, let $u[i] = w_{k'i}$, ini$[i] = k'$.

Algorithm:
(1) Let $P = \{1\}$, $S = \emptyset$.
$\forall j \in V$, let ini$[j] = 1$, $u[j] = w_{1j}$.
(2) Let $u[k] = \min_{j \in V-P} u[j]$.
(3) If degree[ini$[k]$] $\geq b$, then
begin
$\forall j \in V - P$, let $w_{\text{ini}[k],j} = w_{j,\text{ini}[k]} = \infty$.
$\forall j \in V - P$, assume $w_{g'j} = \min_{g \in P} w_{gj}$, let $u[j] = w_{g'j}$, ini$[j] = g'$.
goto (2).
end
(4) Let $P = P \cup \{k\}$, $S = S \cup \{(\text{ini}[k], k), (k, \text{ini}[k])\}$.
(5) If $P \neq V$, then
begin
$\forall j \in V - P$, if $w_{kj} < u[j]$, then let $u[j] = w_{kj}$, ini$[j] = k$.
goto (2).
end
(6) Return S and its cost.

Fig. 5.2.3.1 gen_dcst algorithm

The infoSchema of Fig. 3.5.1 is used in this section to transform the data flow aspect of the gen_dcst algorithm. However, not all of it is needed. The bold area of Fig. 5.2.3.2 gives the relevant part. The infoMap which gives a transformed data flow is shown in Fig. 5.2.3.3.

	A	A	4	(View)
	O	.	x	(Algorithm)
(Declarative Statement)	F	**O**	**14**	**(Data-object/Attribute)**
(Conditional Statement)	.	**F**	**5**	**(preCondition)**
(Imperative Statement)	.	**F**	**15**	**(Action)**
(Procedural Statement)	.	F	x	(Procedure)
(Conditional Statement)	.	F	x	(postCondition)

Fig. 5.2.3.2 infoSchema of the data flow for the gen_dcst a algorithm: bold area

{Declarative Statement}	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	14	*{Attribute/ data-object}*
V	o	.	.	.	.	.	.	.	.	.	.	.	.	.		the set of nodes
n	.	o	.	.	.	.	.	.	.	.	.	.	.	.		no. of nodes in the graph
Ini[]	.	.	o	.	.	.	.	.	.	.	.	.	.	.		Function; a node corresponding to j
j	.	.	.	o	.	.	.	.	.	.	.	.	.	.		a node in V
U[]	.	.	.	.	o	.	.	.	.	.	.	.	.	.		an array of weights
W	.	.	.	.	.	o	.	.	.	.	.	.	.	.		2-dimensional matrix of weights
P	.	.	.	.	.	.	o	.	.	.	.	.	.	.		a subset of nodes
S	.	.	.	.	.	.	.	o	.	.	.	.	.	.		a subset of edges in a graph
k	.	.	.	.	.	.	.	.	o	.	.	.	.	.		a node not in P
b	.	.	.	.	.	.	.	.	.	c	.	.	.	.		max. edges connected to a node
g	.	.	.	.	.	.	.	.	.	.	o	.	.	.		a node of P
g'	.	.	.	.	.	.	.	.	.	.	.	o	.	.		a node not in P
cost of S	.	.	.	.	.	.	.	.	.	.	.	.	o	.		summation of weights in a tree
∞	.	.	.	.	.	.	.	.	.	.	.	.	.	o		two nodes with no edges between them
{Conditional Statement}	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	5	*{preCondition}*
	i	.	.	i	.	.	.	.	.	.	.	.	.	.		j an element of V
degree[ini[k]] >= b	.	.	i	.	.	.	.	.	i	i	.	.	.	.		degree of initial node >= b
	i	.	.	i	.	.	i	.	.	.	.	.	.	.		j an element of V-P
P <> V	i	.	.	.	.	.	i	.	.	.	.	.	.	.		nodes in DCST not equal to V
Wkj < U[j]	.	.	.	i	i	i	.	.	i	.	.	.	.	.		wt. of current edge < wt. of edge ij
{Imperative Statement}	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	15	*{Action}*
P := 1	.	.	.	.	.	.	o	.	.	.	.	.	.	.		set nodes already in DCST to 1
S := 0	.	.	.	.	.	.	.	o	.	.	.	.	.	.		set edges already in DCST to 0
Ini[j] :- 1	.	.	o	i	.	.	.	.	.	.	.	.	.	.		let initial node corresponding to j be 1
U[j] := W1j	.	.	.	i	o	i	.	.	.	.	.	.	.	.		let U[j] be the weight of edge 1j
U[k] = min U[j];	.	.	.	i	b	.	.	.	i	.	.	.	.	.		set weight of k to the min of u[j]
W(ini[k],j) := ∞	.	.	i	i	.	o	.	.	i	.	.	.	.	i		let matrix wts. of nodes not in P be ∞
W(j, ini[k]) := ∞	.	.	i	i	.	o	.	.	i	.	.	.	.	i		let matrix wts. of nodes not in P be ∞
Wg'j:= min(g elem. of P Wgj)	.	.	.	i	.	b	i	.	.	.	i	i	.	.		let wt. of edge not in P be min of corresp edge in P
U[j] := Wg'j	.	.	.	i	o	i	.	.	.	.	.	i	.	.		let U[j] be that wt. not in P
ini[j] := g'	.	.	o	i	.	.	.	.	.	.	.	i	.	.		let node corresponding to j be not in P
P := P U {k}	.	.	.	.	.	.	b	.	i	.	.	.	.	.		update set of nodes in DCST
S := S U {(ini[k],k),(k,ini[k])}	.	.	i	.	.	.	.	b	i	.	.	.	.	.		update set of edges in DCST
U[j] := Wkj	.	.	.	i	o	i	.	.	i	.	.	.	.	.		let U[j] be wt. of current edge kj
ini[j] := k	.	.	o	i	.	.	.	.	i	.	.	.	.	.		let node corresponding to j be k
	.	.	.	.	.	.	.	o	.	.	.	.	o	.		Return S and its cost
{Conditional Statement}	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	.	*{postCondition}*

Fig. 5.2.3.3 Data flow for gen_dcst algorithm

5.2.4 Control flow:

The steps required for the transformation of the "control" aspect of the algorithms are followed. However, steps (h) and (i) are carried out only for the gen_dcst algorithm but not for the others since these two steps are demonstrated in full in section 5.1 and appendix III. Fig. 5.2.4.1 gives the control flow for the gen_dcst algorithm.

Inputs: $G = (V, E)$, $V = \{1, 2, \ldots, n\}$, $b > 0$.
$\forall i, j \in V$, w_{ij} is the weight between i and j.

Definitions: P : set of nodes already in DCST. $P \subseteq V$.
S : set of edges already in DCST. $S \subseteq E$.
$\forall i \in V$, degree[i] is the current degree of i in DCST.
$\forall i \in V$, assume $w_{k'i} = \min_{k \in P} w_{ki}$, let $u[i] = w_{k'i}$, ini[i] $= k'$.

Algorithm:
(1) Let $P = \{1\}$, $S = \emptyset$.
    $\forall j \in V$, let ini[j] $= 1$, $u[j] = w_{1j}$.
(2) Let $u[k] = \min_{j \in V-P} u[j]$.
(3) If degree[ini[k]] $\geq b$, then
    begin
      $\forall j \in V - P$, let $w_{\text{ini}[k],j} = w_{j,\text{ini}[k]} = \infty$.
      $\forall j \in V - P$, assume $w_{g'j} = \min_{g \in P} w_{gj}$, let $u[j] = w_{g'j}$, ini[j] $= g'$.
      goto (2).
    end
(4) Let $P = P \cup \{k\}$, $S = S \cup \{(\text{ini}[k], k), (k, \text{ini}[k])\}$.
(5) If $P \neq V$, then
    begin
      $\forall j \in V - P$, if $w_{kj} < u[j]$, then let $u[j] = w_{kj}$, ini[j] $= k$.
      goto (2).
    end
(6) Return S and its cost.

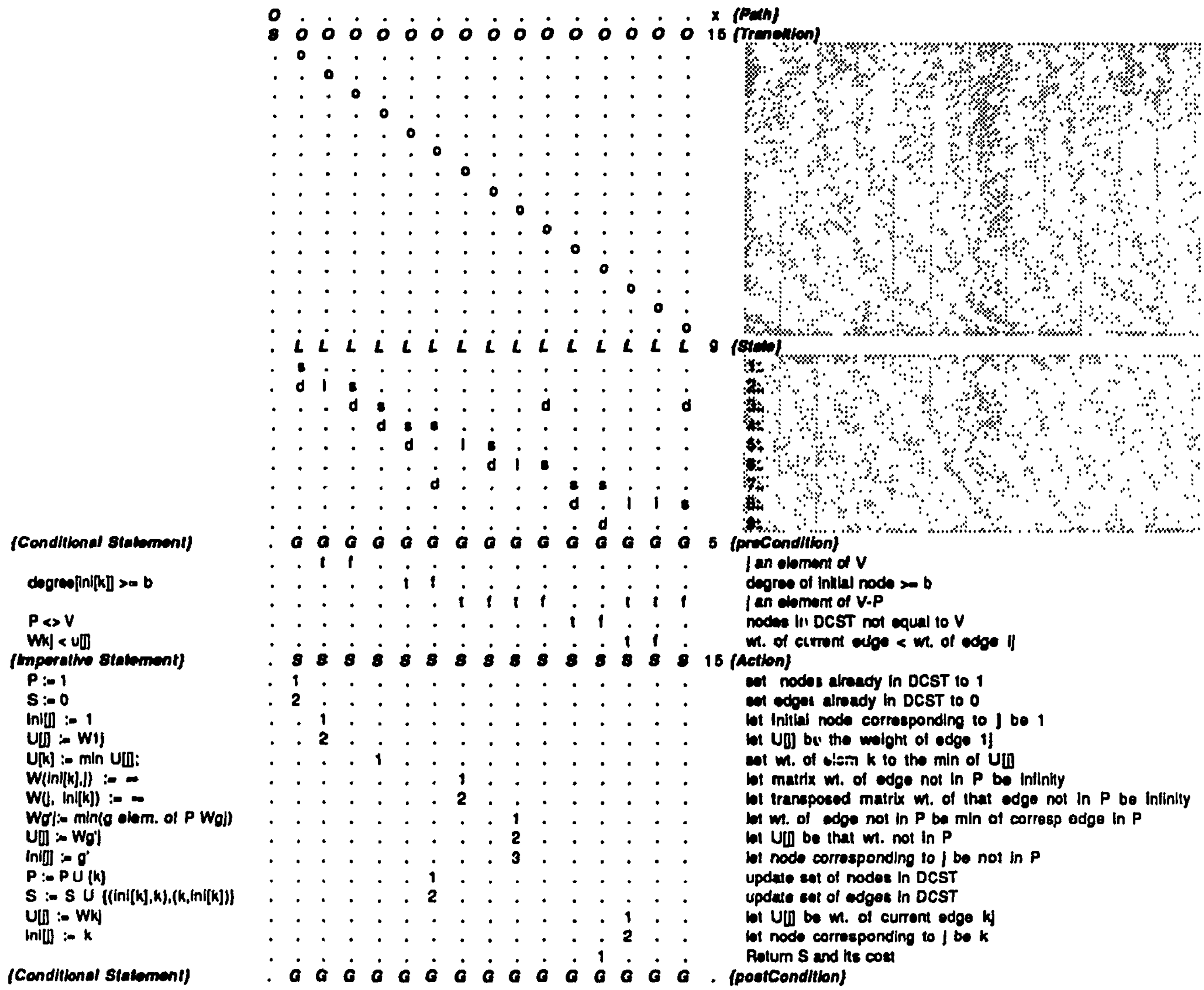
a) source code: original

Inputs: $G = (V, E)$, $V = \{1, 2, \ldots, n\}$, $b > 0$.
$\forall i, j \in V$, w_{ij} is the weight between i and j.

Definitions: P : set of nodes already in DCST. $P \subseteq V$.
S : set of edges already in DCST. $S \subseteq E$.
$\forall i \in V$, degree[i] is the current degree of i in DCST.
$\forall i \in V$, assume $w_{k'i} = \min_{k \in P} w_{ki}$, let $u[i] = w_{k'i}$, ini[i] $= k'$.

Algorithm:
1: Let $P = \{1\}$, $S = \emptyset$.
2: $\forall j \in V$, let ini[j] $= 1$, $u[j] = w_{1j}$.
3: Let $u[k] = \min_{j \in V-P} u[j]$.
4: If degree[ini[k]] $\geq b$, then
  begin
5:   $\forall j \in V - P$, let $w_{\text{ini}[k],j} = w_{j,\text{ini}[k]} = \infty$.
6:   $\forall j \in V - P$, assume $w_{g'j} = \min_{g \in P} w_{gj}$, let $u[j] = w_{g'j}$, ini[j] $= g'$.
   goto 3.
  end
 Let $P = P \cup \{k\}$, $S = S \cup \{(\text{ini}[k], k), (k, \text{ini}[k])\}$.
7: If $P \neq V$, then
  begin
8:   $\forall j \in V - P$, if $w_{kj} < u[j]$, then let $u[j] = w_{kj}$, ini[j] $= k$.
   goto 3.
  end
9: Return S and its cost.

b) edited source code: line nos removed, states 1..9 identified



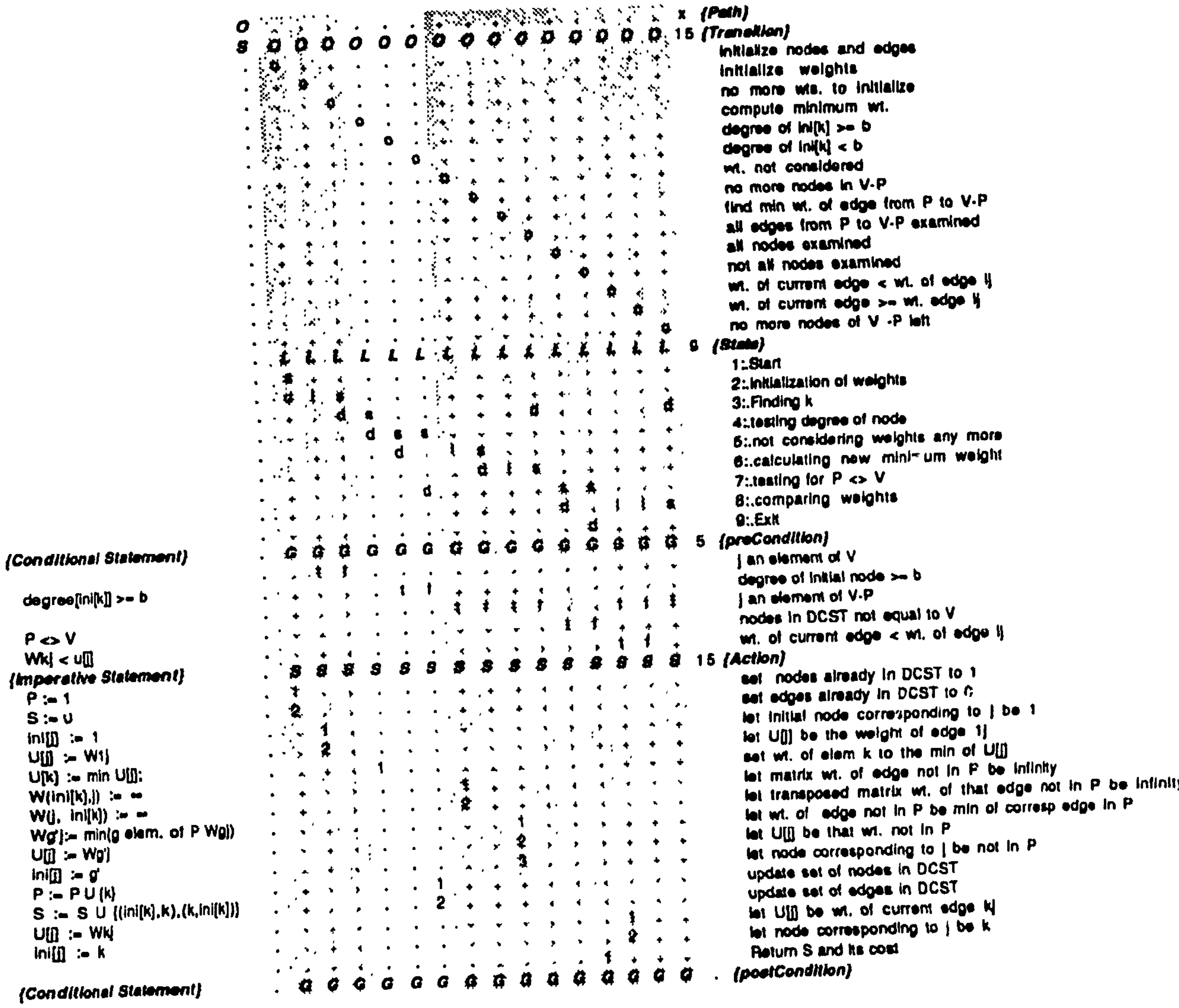
c) Normalized infoMap: shaded areas to be filled in

	O	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	*{Path}*
	S	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	15	*{Transition}*
	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.		initialize nodes and edges
	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.		initialize weights
	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.		no more wts. to initialize
	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.		compute minimum wt.
	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		degree of ini[k] >= b
	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.		degree of ini[k] < b
	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.		wt. not considered
	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.		no more nodes in V-P
	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.		find min wt. of edge from P to V-P
	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.		all edges from P to V-P examined
	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.		all nodes examined
	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.		not all nodes examined
	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.		wt. of current edge < wt. of edge ij
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.		wt. of current edge >= wt. edge ij
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		no more nodes of V-P left
	.	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	9	*{State}*
	.	s	.	.	.	.	.	.	.	.	.	.	.	.	.	.		1:Start
	.	d	l	s	.	.	.	.	.	.	.	.	.	.	.	.		2:Initialization of weights
	.	.	.	d	s	.	.	.	.	.	d	.	.	.	.	d		3:Finding k
	.	.	.	.	d	s	s	.	.	.	.	.	.	.	.	.		4:testing degree of node
	.	.	.	.	.	d	.	l	s	.	.	.	.	.	.	.		5:not considering weights any more
	.	.	.	.	.	.	.	.	d	l	s	.	.	.	.	.		6:calculating new minimum weight
	.	.	.	.	.	.	d	.	.	.	.	s	s	.	.	.		7:testing for P <> V
	.	.	.	.	.	.	.	.	.	.	.	d	.	l	l	s		8:comparing weights
	.	.	.	.	.	.	.	.	.	.	.	.	d	.	.	.		9:Exit
{Conditional Statement}	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	5	*{preCondition}*
	.	.	t	f	.	.	.	.	.	.	.	.	.	.	.	.		j an element of V
degree[ini[k]] >= b	.	.	.	.	.	t	f	.	.	.	.	.	.	.	.	.		degree of initial node >= b
	.	.	.	.	.	.	.	t	f	t	f	.	.	t	t	f		j an element of V-P
P <> V	.	.	.	.	.	.	.	.	.	.	.	t	f	.	.	.		nodes in DCST not equal to V
Wkj < u[j]	.	.	.	.	.	.	.	.	.	.	.	.	.	t	f	.		wt. of current edge < wt. of edge ij
{Imperative Statement}	.	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	15	*{Action}*
P := 1	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.		set nodes already in DCST to 1
S := 0	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.		set edges already in DCST to 0
ini[j] := 1	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.		let initial node corresponding to j be 1
U[j] := W1j	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.		let U[j] be the weight of edge 1j
U[k] := min U[j];	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.		set wt. of elem k to the min of U[j]
W(ini[k],j) := ∞	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.		let matrix wt. of edge not in P be infinity
W(j, ini[k]) := ∞	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.	.		let transposed matrix wt. of that edge not in P be infinity
Wg'j:= min(g elem. of P Wgj)	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.		let wt. of edge not in P be min of corresp edge in P
U[j] := Wg'j	.	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.		let U[j] be that wt. not in P
ini[j] := g'	.	.	.	.	.	.	.	.	.	3	.	.	.	.	.	.		let node corresponding to j be not in P
P := P U {k}	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.		update set of nodes in DCST
S := S U {(ini[k],k),(k,ini[k])}	.	.	.	.	.	.	2	.	.	.	.	.	.	.	.	.		update set of edges in DCST
U[j] := Wkj	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.		let U[j] be wt. of current edge kj
ini[j] := k	.	.	.	.	.	.	.	.	.	.	.	.	.	2	.	.		let node corresponding to j be k
	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.		Return S and its cost
{Conditional Statement}	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	.	*{postCondition}*

d) Normalized infoMap: states and transitions added (shaded area)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
																X	{Path}
	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	15	{Transition}
	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.		initialize nodes and edges
	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.		initialize weights
	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.		no more wts. to initialize
	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.		compute minimum wt.
	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		degree of ini[k] >= b
	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.		degree of ini[k] < b
	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.		wt. not considered
	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.		no more nodes in V-P
	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.		find min wt. of edge from P to V-P
	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.		all edges from P to V-P examined
	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.		all nodes examined
	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.		not all nodes examined
	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.		wt. of current edge < wt. of edge ij
	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.		wt. of current edge >= wt. edge ij
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		no more nodes of V -P left
	L	L	L	L	L	L	L	L	L		L	L	L	L	L	9	{State}
	s	.	.	.	.	.	.	.	.	.	.	.	.	.	.		1:.Start
	d	l	s	.	.	.	.	.	.	.	.	.	.	.	.		2:.Initialization of weights
	.	.	d	s	.	.	.	.	.	d	.	.	.	.	d		3:.Finding k
	.	.	.	d	s	s	.	.	.	.	.	.	.	.	.		4:.testing degree of node
	.	.	.	.	d	.	l	s	.	.	.	.	.	.	.		5:.not considering weights any more
	.	.	.	.	.	.	.	d	l	s	.	.	.	.	.		6:.calculating new minimum weight
	.	.	.	.	.	d	.	.	.	.	s	s	.	.	.		7:.testing for P <> V
	.	.	.	.	.	.	.	.	.	.	d	.	l	l	s		8:.comparing weights
	.	.	.	.	.	.	.	.	.	.	.	d	.	.	.		9:.Exit
{Conditional Statement}	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	5	*{preCondition}*
		t	f														j an element of V
degree[ini[k]] >= b					t	f											degree of initial node >= b
							t	f	t	f			t	t	f		j an element of V-P
P <> V											t	f					nodes in DCST not equal to V
Wkj < U[j]													t	f			wt. of current edge < wt. of edge ij
{Imperative Statement}	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	15	*{Action}*
P := 1	1																set nodes already in DCST to 1
S := 0	2																set edges already in DCST to 0
ini[j] := 1		1															let initial node corresponding to j be 1
U[j] := W1j		2															let U[j] be the weight of edge 1j
U[k] := min U[j];				1													set wt. of elem k to the min of U[j]
W(ini[k],j) := ∞							1										let matrix wt. of edge not in P be infinity
W(j, ini[k]) := ∞							2										let transposed matrix wt. of that edge not in P be infinity
Wg' := min(g elem. of P Wgj)									1								let wt. of edge not in P be min of corresp edge in P
U[j] := Wg'									2								let U[j] be that wt. not in P
ini[j] := g'									3								let node corresponding to j be not in P
P := P U {k}						1											update set of nodes in DCST
S := S U {(ini[k],k),(k,ini[k])}						2											update set of edges in DCST
U[j] := Wkj													1				let U[j] be wt. of current edge kj
ini[j] := k													2				let node corresponding to j be k
												1					Return S and its cost
{Conditional Statement}	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	.	*{postCondition}*

e) Normalized infomap: state diagram (unshaded area)



f) Normalized infoMap to be optimized: shaded areas

	O	.	.	.	.	.	.	.	.	.	.	.	x	*{Path}*
	S	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	11	*{Transition}*
	.	o	.	.	.	.	.	.	.	.	.	.		Initialize weights
	.	.	o	.	.	.	.	.	.	.	.	.		Initialize nodes and edges
	.	.	.	o	.	.	.	.	.	.	.	.		compute minimum wt.
	.	.	.	.	o	.	.	.	.	.	.	.		degree of ini[k] >= b
	.	.	.	.	.	o	.	.	.	.	.	.		degree of ini[k] < b
	.	.	.	.	.	.	o	.	.	.	.	.		wt. not considered
	.	.	.	.	.	.	.	o	.	.	.	.		find min wt. of edge from P to V-P
	.	.	.	.	.	.	.	.	o	.	.	.		weight of current edge < wt. of edge ij
	.	.	.	.	.	.	.	.	.	o	.	.		weight of current edge >= wt.of edge ij
	.	.	.	.	.	.	.	.	.	.	o	.		no more nodes of V-P left
	.	.	.	.	.	.	.	.	.	.	.	o		all nodes in V examined
	.	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	7	*{State}*
	.	l	s	.	.	.	.	.	.	.	.	.		1:.Start initialization of weights
	.	.	d	s	.	.	.	e	.	.	d	.		2:.Finding k
	.	.	.	d	s	s	.	.	.	.	.	.		3:.testing degree of node
	.	.	.	.	d	.	l	.	.	.	.	.		4:.not considering weights any more
	.	.	.	.	.	.	e	l	.	.	.	.		5:.calculating new minimum weight
	.	.	.	.	.	d	.	.	l	l	s	s		6:.comparing weights
	.	.	.	.	.	.	.	.	.	.	.	d		7:.Exit
{Conditional Statement}	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	5	*{preCondition}*
j an element of V	.	t	f	.	.	.	.	.	.	.	.	.		j an element of V
degree[ini[k]] >= b	.	.	.	.	t	f	.	.	.	.	.	.		degree of initial node >= b
P <> V	.	.	.	.	.	.	.	.	t	t	t	f		nodes in DCST not equal to V
j an element of V-P	.	.	.	.	.	.	.	.	t	t	f	.		j an element of V-P
Wkj < u[j]	.	.	.	.	.	.	.	.	t	f	.	.		wt. of current edge < wt. of edge ij
{Imperative Statement}	.	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	15	*{Action}*
ini[j] := 1	.	1	.	.	.	.	.	.	.	.	.	.		let initial node corresponding to j be 1
U[j] := W1j	.	2	.	.	.	.	.	.	.	.	.	.		let U[j] be the weight of edge 1j
P := 1	.	.	1	.	.	.	.	.	.	.	.	.		set nodes already in DCST to 1
S := 0	.	.	2	.	.	.	.	.	.	.	.	.		set edges already in DCST to 0
U[k] := min U[j];	.	.	.	1	.	.	.	.	.	.	.	.		set wt. of elem k to the min of U[j]
P := P U {k}	.	.	.	.	.	1	.	.	.	.	.	.		update set of nodes in DCST
S := S U {(ini[k],k),(k,ini[k])}	.	.	.	.	.	2	.	.	.	.	.	.		update set of edges in DCST
W(ini[k],j) := ∞	.	.	.	.	.	.	1	.	.	.	.	.		let matrix wt. of edge not in P be infinity
W(j, ini[k]) := ∞	.	.	.	.	.	.	2	.	.	.	.	.		let transposed matrix wt. of that edge not in P be infinity
Wg'j := min(g elem. of P Wgj)	.	.	.	.	.	.	.	1	.	.	.	.		let wt. of edge not in P be min of corresp edge in P
U[j] := Wg'j	.	.	.	.	.	.	.	2	.	.	.	.		let U[j] be that wt. not in P
ini[j] := g'	.	.	.	.	.	.	.	3	.	.	.	.		let node corresponding to j be not in P
U[j] := Wkj	.	.	.	.	.	.	.	.	1	.	.	.		let U[j] be wt. of current edge kj
ini[j] := k	.	.	.	.	.	.	.	.	2	.	.	.		let node corresponding to j be k
	.	.	.	.	.	.	.	.	.	.	.	1		Return S and its cost
{Conditional Statement}	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	1	*{postCondition}*
j an element of V-P	.	.	.	.	.	.	t	t	.	.	.	.		j an element of V-P

g) Optimized and documented infoMap

1:. *Start initialization of weights*
initialize weights, CONTINUE AT 1
initialize nodes and edges, CONTINUE AT 2

2:. *Finding k*
compute minimum wt., CONTINUE AT 3

3:. *testing degree of node*
degree of ini[k] >= b, CONTINUE AT 4
degree of ini[k] < b, CONTINUE AT 6

4:. *not considering weights any more*
wt. not considered, CONTINUE AT 4
exit, CONTINUE AT 5

5:. *calculating new minimum weight*
find min wt. of edge from P to V-P, CONTINUE AT 5
exit, CONTINUE AT 2

6:. *comparing weights*
weight of current edge < wt. of edge ij, CONTINUE AT 6
weight of current edge >= wt. of edge ij, CONTINUE AT 6
no more nodes of V-P left, CONTINUE AT 2
all nodes in V examined, CONTINUE AT 7

7:. *Exit*

h) Optimized state diagram as Structured English (generated)

1:. *Start initialization of weights*
initialize weights, CONTINUE AT 1
j elem. of V -> ini[j] := 1; U[j] := Wij; goto 1

initialize nodes and edges, CONTINUE AT 2
j not elem. of V -> P:= 1; S:= 0; goto 2

2:. *Finding k*
compute minimum wt., CONTINUE AT 3
U[k] := min U[j]; goto 3

3:. *testing degree of node*
degree of ini[k] >= b, CONTINUE AT 4
degree[ini[k]] >= b -> goto 4

degree of ini[k] < b, CONTINUE AT 6
degree[ini[k]] < b -> P := P U {k}; S := S U {(ini[k],k)),(k,ini[k])}; goto 6

4:. *not considering weights any more*
wt. not considered, CONTINUE AT 4
j elem. of V-P -> W(ini[k],j) := ∞; W(j,ini[k]) := ∞; goto 4

exit, CONTINUE AT 5
j not elem. of V-P -> goto 5

5:. *calculating new minimum weight*
find min wt. of edge from P to V-P, CONTINUE AT 5
j elem. of V-P -> Wg'j:= min(g elem. of P Wgj); U[j]:= Wg'j; ini[j]:= g'; goto 5

exit, CONTINUE AT 2
j not elem. of V-P -> goto 2

6:. *comparing weights*
weight of current edge < wt. of edge ij, CONTINUE AT 6
P <> V and j elem. of V-P and Wkj < U[j] -> U[j] := Wkj; ini[j] := k; goto 6

weight of current edge >= wt. of edge ij, CONTINUE AT 6
P <> V and j elem. of V-P and Wkj >= U[j] -> goto 6

no more nodes of V-P left, CONTINUE AT 2
P <> V and j not elem. of V-P -> goto 2

all nodes in V examined, CONTINUE AT 7
P = V -> Return S and its cost; goto 7

7:. *Exit*

i) source code (generated)

Fig. 5.2.4.1 Control flow for GEN_DCST algorithm

5.2.5 Summary

The algorithm used in section 5.2 shows a reduction in the number of transactions (from 15 to 11) and the number of states (from 9 to 7). These demonstrate the effectiveness of TAPi in reducing complexity.

5.3 Skeletonization of Binary patterns algorithm

5.3.1 Introduction

The TAPi methodology is used once more for the transformation of the Skeletonization of Binary patterns algorithm. The algorithm consists of two procedures and two functions. A detailed description of these is given in [4]. However, a narrative problem statement is presented below and reads as follows:

"Given an original pattern, change dark points along its edges to white points until the pattern is thinned to a line drawing. Retain connectedness and shape of the original pattern".

This problem statement is represented in infoMap as shown in Fig. 5.3.1.1.

A C {Problem}
v   of skeletonizing binary patterns
S 4 {Sentence}
1   Given an original pattern; change dark points along
2   its edges to white points until the pattern is thinned
3   to a line drawing. Retain connectedness and shape of
4   the original pattern.

Fig. 5.3.1.1 Skeletonization of Binary patterns Problem Statement (Narrative)

The infoSchema for the skeletonization algorithm is given in Fig. 5.3.1.2. The area shaded indicates the additions to Fig. 3.2.1. The infoSchema in the hierarchical view of Fig. 5.3.1.2 represents a tree of calls, in that the SPTA procedure invokes SKELETONIZE() which in turn invokes functions EDGEPOINT and SAFEPOINT.

```
                                A A A A A A A A  4 {View}
                                v v . . . . . .    1:. Data Model
                                . . v v . . . .    2:. Hierarchy
                                . . . . v v . .    3:. Data Flow
                                . . . . . . v v    4:. Control Flow
{Procedural Statement}          O . H H O . . .  4 {Procedure/Function}
  ONE_PROCESSOR_SPTA()          . . h . . . . .    ONE_PROCESSOR_SPTA
  SKELETONIZE()                 . . 1 h . . . .    SKELETONIZE
  EDGEPOINT()                   . . . 1 . . . .    EDGEPOINT
  SAFEPOINT()                   . . . 2 . . . .    SAFEPOINT
                                . O . . . . . .  4 {Type}
{Declarative Statement}         M M . . F O . . 18 {Attribute}
                                . . . . . . O .  x {Path}
                                . . . . . F S O 20 {Transition}
                                . . . . . F . L 11 {State}
{Conditional Statement}         . . . . . F . G 14 {preCondition}
{Imperative Statement}          . . . . . F . S 18 {Action}
{Conditional Statement}         . . . . . F . G  x {postCondition}
```

Fig. 5.3.1.2 infoSchema for the Skeletonization of binary patterns

5.3.2 Data model and data flow

A data model is built for the procedures and functions in the skeletonization algorithm. This is shown in Fig. 5.3.2.1. The data flow for this algorithm is presented in Fig. 5.3.2.2.

	O	O	O	O	.	.	.	.	.	4	*(Procedure/Function)*
	o	.	.	.	.	.	.	.	.		ONE_PROCESSOR_SPTA
	.	o	.	.	.	.	.	.	.		Skeletonization
	.	.	o	.	.	.	.	.	.		Edgepoint
	.	.	.	o	.	.	.	.	.		Safepoint
(Declarative Statement)	.	.	.	.	O	O	O	O	O	5	*(DataType)*
integer	.	.	.	.	o	.	.	.	.		integer
count-type	.	.	.	.	.	o	.	.	.		count-type
pat-type	.	.	.	.	.	.	o	.	.		pat-type
border-type	.	.	.	.	.	.	.	o	.		border-type
	.	.	.	.	.	.	.	.	o		array
(Declarative Statement)	M	M	M	M	M	M	M	M	M	18	*(Attribute/data-object)*
i;	a	a	.	.	v	.	.	.	.		pass no. or iteration number
d;	a	a	.	.	.	v	.	.	v		array containing the no. of dark points
k;	a	.	.	.	v	.	.	.	.		scan number
j;	a	a	a	.	v	.	.	.	.		scan type
pattern;	a	.	.	.	v	.	v	.	.		pattern
MAXSCAN	a	.	.	.	v	.	.	.	.		maximum scan number
MAXROW	a	.	.	.	v	.	.	.	.		total no. of rows in pattern
row;	.	a	.	.	v	.	.	.	.		row
column;	.	a	.	.	v	.	.	.	.		column
P	.	a	.	.	v	.	.	.	.		point co-ordinates
n;	.	a	a	a	.	.	.	.	v		array [0 .. 7] of pointers
border;	.	a	a	a	v	.	.	v	.		border
first-row;	.	a	.	.	v	.	.	.	.		first row
last-row;	.	a	.	.	v	.	.	.	.		last row
EDGEPOINT()	.	a	a	.	v	.	.	.	.		EDGEPOINT function
SAFEPOINT()	.	a	.	a	v	.	.	.	.		SAFEPOINT function
ADJUST()	.	a	.	.	v	.	.	.	.		ADJUST function
MAXINT	.	a	.	.	v	.	.	.	.		a white point

Fig. 5.3.2.1 Data model for the skeletonization of binary patterns algorithm

	O	O	O	O	4	*(Procedure/Function)*
	o	.	.	.		SPTA
	.	o	.	.		Skeletonization
	.	.	o	.		Edgepoint
	.	.	.	o		Safepoint
(Declarative Statement)	F	F	F	F	18	*(Attribute/ data-object)*
i;	b	i	.	.		pass no. or iteration number
d;	b	b	.	.		array containing the no. of dark points
k;	b	.	.	.		scan number
j;	b	i	i	.		scan type
pattern;	i	.	.	.		pattern
MAXSCAN	i	.	.	.		maximum scan number
MAXROW	i	.	.	.		total no. of rows in pattern
row;	.	o	.	.		row
column;	.	b	.	.		column
P	.	b	.	.		point co-ordinates
n;	.	i	i	i		array [0 .. 7] of pointers
border;	.	i	o	i		border
first-row;	.	i	.	.		first row
last-row;	.	i	.	.		last row
EDGEPOINT()	.	i	o	.		EDGEPOINT function
SAFEPOINT()	.	i	.	o		SAFEPOINT function
ADJUST()	.	i	.	.		ADJUST function
MAXINT	.	i	.	.		a white point

Fig. 5.3.2.2 Data flow for the skeletonization of binary patterns algorithm

5.3.3 Recreating ONE_PROCESSOR_SPTA procedure using comments only

Conventional programming languages or development tools are not highly expressive [50] and thus, limit one's ability to document algorithms properly. To demonstrate this, an attempt is made to recreate the ONE_PROCESSOR_SPTA procedure using only its comments. This recreation process is illustrated in Fig. 5.3.3.1.

Fig. 5.3.3.1 (a) shows the comments in a sequential order. States are identified in (b). It is noticeable that states of Fig. 5.3.3.1 (b) does not match those of Fig. 5.3.4.1 (b). In the normalized infoMap, Fig. 5.3.3.1 (c), it is difficult to determine where to continue the repeat. However, using the comments in conjunction with the code, Fig. 5.3.4.1 (b), there appears to be no difficulty in determining where to continue the repeat.

S	{Comment}
1	Initialize the pass number
2	Initialize d to ZERO for each scan
3	Initialize the scan number
4	Increment the pass number by one
5	Set scan type to left/right edgepoints
6	Increment the scan number by one
7	Execute the kth scan on the entire pattern
8	If criterion1 and criterion2 are FALSE then prepare for the next scan
9	Set scan type to top/bottom edgepoints
10	Increment the scan number by one
11	Execute the kth scan on the entire pattern
12	Repeat executing passes on the entire pattern until criterion1 or criterion2 is TRUE

a) comments of procedure: original

1: Initialize the pass number

2: Initialize d to ZERO for each scan
Initialize the scan number
Increment the pass number by one
Set scan type to left/right edgepcints
Increment the scan number by one
Execute the kth scan on the entire pattern

3: If criterion1 and criterion2 are FALSE
then prepare for the next scan
Set scan type to top/bottom edgepoints
Increment the scan number by one
Execute the kth scan on the entire pattern

4: Repeat executing passes on the entire pattern
until criterion1 or criterion2 is TRUE

5:

b) edited comments of procedure: states 1 .. 5 identified

	o	.	.	.	.	.	.	.	x	*(Path)*
	s	o	o	o	o	o	o	o	7	*(Transition)*
	.	o	.	.	.	.	.	.		
	.	.	o	.	.	.	.	.		
	.	.	.	o	.	.	.	.		
	.	.	.	.	o	.	.	.		
	.	.	.	.	.	o	.	.		
	.	.	.	.	.	.	o	.		
	.	.	.	.	.	.	.	o		
	.	*L*	*L*	*L*	*L*	*L*	*L*	*L*	5	*(State)*
	.	s	.	.	.	.	.	.		1:
	.	d	l	s	.	.	.	.		2:
	.	.	.	d	s	s	d	.		3:
	.	.	.	.	d	d	s	s		4:
	.	.	.	.	.	.	.	d		5:
(Conditional Statement)	.	G	G	G	G	G	G	G	2	*(preCondition)*
	.	.	.	.	t	f	.	.		criterion1 and criterion2
	.	.	.	.	.	.	f	t		criterion1 or criterion2
(Imperative Statement)	.	S	S	S	S	S	S	S	9	*(Action)*
	.	1	.	.	.	.	.	.		initialize the pass number
	.	.	1	.	.	.	.	.		initialize d to ZERO
	.	.	.	1	.	.	.	.		initialize the scan number
	.	.	.	.	1	.	.	.		set scan type to top/bottom edgepoints
	.	.	.	.	.	1	.	.		prepare for the next scan
	.	.	.	2	2	.	.	.		increment the pass number by one
	.	.	.	3	.	.	.	.		set scan type to left/right edgepoints
	.	.	.	4	.	.	.	.		increment the scan number by one
	.	.	.	5	3	.	.	.		execute the kth scan on the entire pattern
(Conditional Statement)	.	G	G	G	G	G	G	G	.	*(postCondition)*

c) Normalized infoMap: shaded area to be filled in

Fig. 5.3.3.1 Recreating ONE_PROCESSOR_SPTA using comments only

5.3.4 Control flow:

Using both comments and code of the skeletonization algorithm, its "control" aspect is transformed by following the steps stated in chapter 3. Each step - (a) to (i) -is carried out for the ONE_PROCESSOR_SPTA procedure; but for the other procedure and two functions, steps (h) and (i) are omitted. These are crucial steps and therefore, should not be omitted. Since these nine steps have been demonstrated in Fig. 5.3.4.1 and elsewhere in this thesis, therefore, there is no need for repetitions.

The transformed control flow for the procedure mentioned above is shown in Fig. 5.3.4.1 of this section; the other procedure and two functions are shown in Fig. 5.3.4.2 to Fig. 5.3.4.4 of appendix V.

```
procedure ONE_PROCESSOR_SPTA(var PATTERN : pat_type);

var
     j : integer;          {indicates the type of scan,
                            j = 0   for LR-scan, and
                            j = 2   for TB-scan.}
     k : integer;          {contains the scan number.}
     d: count_type;        {an array containing the number of dark
                            points which have neither been flagged nor
                            declared to be safepoints, remaining in the
                            pattern upon the completion of a scan.}
begin
     i := 0;               {Initialize the pass number.}

     for k := -1 to MAXSCAN do
            d[k] := 0;     {Initialize d to ZERO for each scan.}

     k := 0;               {Initialize the scan number.}

     repeat
```

```
            i := i + 1;     {Increment the pass number by one.}
            j := 0;         {Set scan type to left/right edgepoints.}
            k := k + 1;     {Increment the scan number by one.}

            SKELETONIZE(PATTERN,j,1,MAXROW,d[k]);
                            {Execute the kth scan on the entire pattern.}

            if (d[k] <> 0) and (d[k] <> d[k-2]) then
                            {If criterion1 and criterion2 are FALSE,
                            then prepare for the next scan.}
            begin
            j := 2;         {Set scan type to top/bottom edgepoints.}
            k := k+1;       {Increment the scan number by one.}

            SKELETONIZE(PATTERN,j,1,MAXROW,d[k]);
                            {Execute the kth scan on the entire pattern.}
            end;

        until (d[k] = 0) or (d[k] = d[k-2]);
                            {Repeat executing passes on the entire pattern
                            until criterion1 or criterion2 is TRUE.}
    end;
```

a) source code for procedure ONE_PROCESSOR_SPTA: original

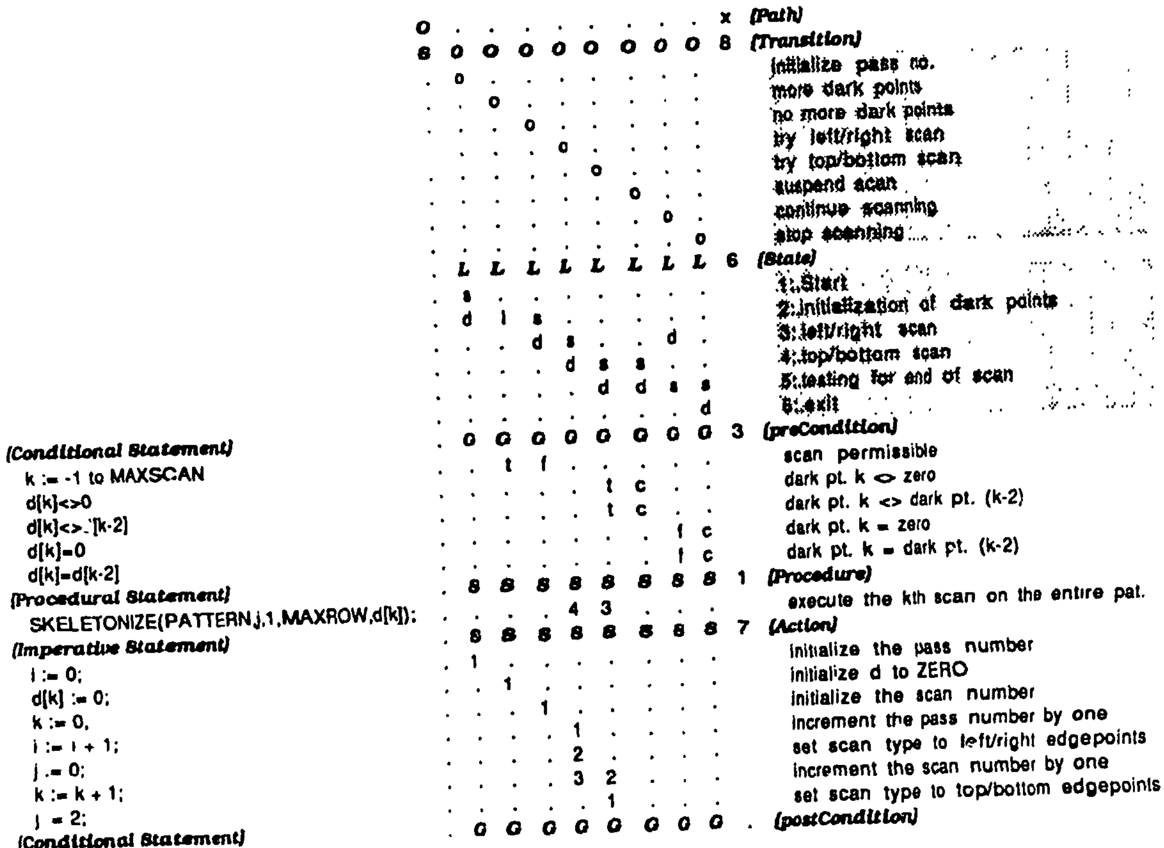
```
    procedure ONE_PROCESSOR_SPTA(var PATTERN : pat_type);

    var
        j : integer;        {indicates the type of scan,
                            j = 0   for LR-scan, and
                            j = 2   for TB-scan.}
        k : integer;        {contains the scan number.}
        d: count_type;      {an array containing the number of dark
                            points which have neither been flagged nor
                            declared to be safepoints, remaining in the
                            pattern upon the completion of a scan.}
     begin
1:      i := 0;             {Initialize the pass number.}

2:      for k := -1 to MAXSCAN do
            d[k] := 0;      {Initialize d to ZERO for each scan.}
```

```
            k := 0;               {Initialize the scan number.}

3:          repeat
                 i := i + 1;      {Increment the pass number by one.}
                 j := 0;          {Set scan type to left/right edgepoints.}
                 k := k + 1;      {Increment the scan number by one.}

                 SKELETONIZE(PATTERN,j,1,MAXROW,d[k]);
                                  {Execute the kth scan on the entire pattern.}

4:               if (d[k] <> 0) and (d[k] <> d[k-2]) then
                                  {If criterion1 and criterion2 are FALSE,
                                  then prepare for the next scan.}
                 begin
                 j := 2;          {Set scan type to top/bottom edgepoints.}
                 k := k+1;        {Increment the scan number by one.}

                 SKELETONIZE(PATTERN,j,1,MAXROW,d[k]);
                                  {Execute the kth scan on the entire pattern.}
                 end;

5:            until (d[k] = 0) or (d[k] = d[k-2]);
                                  {Repeat executing passes on the entire pattern
                                  until criterion1 or criterion2 is TRUE.}
        end;
6:
```

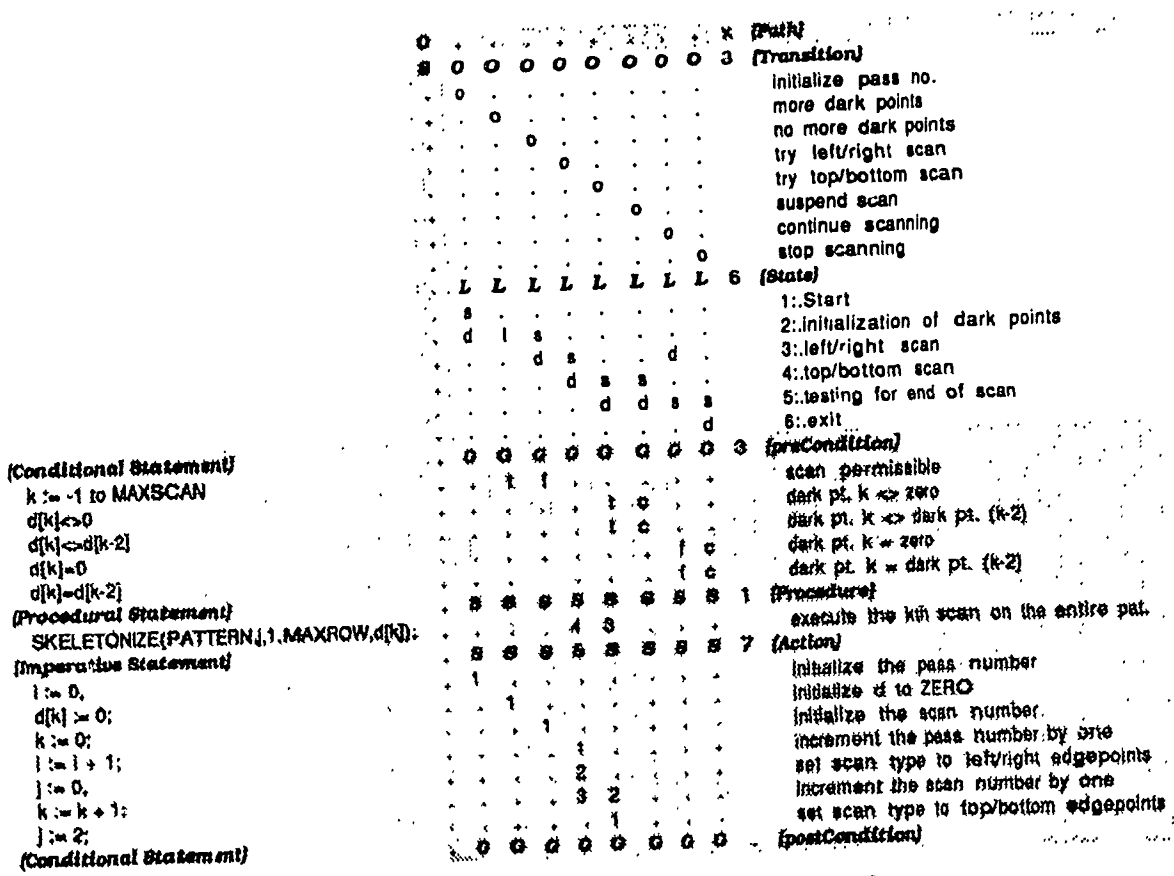
b) edited source code: states 1 .. 6 identified

	O	.	.	.	.	.	.	.	.	x	***{Path}***
	S	O	O	O	O	O	O	O	O	8	***{Transition}***
	.	o	.	.	.	.	.	.	.		
	.	.	o	.	.	.	.	.	.		
	.	.	.	o	.	.	.	.	.		
	.	.	.	.	o	.	.	.	.		
	.	.	.	.	.	o	.	.	.		
	.	.	.	.	.	.	o	.	.		
	.	.	.	.	.	.	.	o	.		
	.	.	.	.	.	.	.	.	o		
	.	L	L	L	L	L	L	L	L	6	***{State}***
	.	s	.	.	.	.	.	.	.		1:
	.	d	l	s	.	.	.	.	.		2:
	.	.	.	d	s	.	.	d	.		3:
	.	.	.	.	d	s	s	.	.		4:
	.	.	.	.	.	d	d	s	s		5:
	.	.	.	.	.	.	.	.	d		6:
{Conditional Statement}	.	G	G	G	G	G	G	G	G	3	***{preCondition}***
k := -1 to MAXSCAN	.	.	t	f	.	.	.	.	.		scan permissible
d[k]<>0	.	.	.	.	.	t	c	.	.		dark pt. k <> zero
d[k]<>d[k-2]	.	.	.	.	.	t	c	.	.		dark pt. k <> dark pt. (k-2)
d[k]=0	.	.	.	.	.	.	.	f	c		dark pt. k = zero
d[k]=d[k-2]	.	.	.	.	.	.	.	f	c		dark pt. k = dark pt. (k-2)
{Procedural Statement}	.	S	S	S	S	S	S	S	S	1	***{Procedure}***
SKELETONIZE(PATTERN,j,1,MAXROW,d[k]);	.	.	.	.	4	3	.	.	.		execute the kth scan on the entire pat
{Imperative Statement}	.	S	S	S	S	S	S	S	S	7	***{Action}***
i := 0;	.	1	.	.	.	.	.	.	.		initialize the pass number
d[k] := 0;	.	.	1	.	.	.	.	.	.		initialize d to ZERO
k := 0;	.	.	.	1	.	.	.	.	.		initialize the scan number
i := i + 1;	.	.	.	.	1	.	.		.		increment the pass number by one
j := 0;	.	.	.	.	2	.	.		.		set scan type to left/right edgepoints
k := k + 1;	.	.	.	.	3	2	.	.	.		increment the scan number by one
j := 2;	.	.	.	.	.	1	.	.	.		set scan type to top/bottom edgepoints
{Conditional Statement}	.	G	G	G	G	G	G	G	G	.	***{postCondition}***

c) Normalized infoMap: shaded areas to be filled in



d) Normalized infoMap: states and transitions added (shaded area)



e) Normalized infoMap: state diagram (unshaded area)

	O	.	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	O	8	*(Transition)*
	.	o	.	.	.	.	.	.	.		initialize pass no.
	.	.	o	.	.	.	.	.	.		more dark points
	.	.	.	o	.	.	.	.	.		no more dark points
	.	.	.	.	o	.	.	.	.		try left/right scan
	.	.	.	.	.	o	.	.	.		try top/bottom scan
	.	.	.	.	.	.	o	.	.		suspend scan
	.	.	.	.	.	.	.	o	.		continue scanning
	.	.	.	.	.	.	.	.	o		stop scanning
	.	L	L	L	L	L	L	L	L	6	*(State)*
	.	s	.	.	.	.	.	.	.		1:.Start
	.	d	s	s	.	.	.	.	.		2:.initialization of dark points
	.	.	.	d	s	.	.	d	.		3:.left/right scan
	.	.	.	.	d	s	s	.	.		4:.top/bottom scan
	.	.	.	.	.	d	d	s	s		5:.testing for end of scan
	.	.	.	.	.	.	.	.	d		6:.exit
(Conditional Statement)	.	G	G	G	G	G	G	G	G	3	*(preCondition)*
k := -1 to MAXSCAN	.	.	t	f	.	.	.	.	.		scan permissible
d[k]<>0	.	.	.	.	.	t	c	.	.		dark pt. k <> zero
d[k]<>d[k-2]	.	.	.	.	.	t	c	.	.		dark pt. k <> dark pt (k-2)
d[k]=0	.	.	.	.	.	.	.	f	c		dark pt. k = zero
d[k]=d[k-2]	.	.	.	.	.	.	.	f	c		dark pt k = dark pt (k-2)
(Procedural Statement)	.	S	S	S	S	S	S	S	S	1	*(Procedure)*
SKELETONIZE(PATTERN,j,1,MAXROW,d[k]);	.	.	.	.	4	3	.	.	.		execute the kth scan on the entire pat.
(Imperative Statement)	.	S	S	S	S	S	S	S	S	7	*(Action)*
i := 0;	.	1	.	.	.	.	.	.	.		initialize the pass number
d[k] := 0;	.	.	1	.	.	.	.	.	.		initialize d to ZERO
k := 0;	.	.	.	1	.	.	.	.	.		initialize the scan number
i := i + 1;	.	.	.	.	1	.	.	.	.		increment the pass number by one
j := 0;	.	.	.	.	2	.	.	.	.		set scan type to left/right edgepoints
k := k + 1;	.	.	.	.	3	2	.	.	.		increment the scan number by one
j := 2;	.	.	.	.	.	1	.	.	.		set scan type to top/bottom edgepoints
(Conditional Statement)	.	G	G	G	G	G	G	G	G		*(postCondition)*

f) Normalized infoMap to be optimized: shaded areas

	O	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	5	*(Transition)*
	.	o	.	.	.	.		more dark points
	.	.	o	.	.	.		no more dark points
	.	.	.	o	.	.		try left/right scan
	.	.	.	.	o	.		try top/bottom scan
	.	.	.	.	.	o		stop scanning
	.	L	L	L	L	L	4	*(State)*
		l	s	.	.	.		1:.start initialization of dark points
	.	.	d	s	d	.		2:.left/right scan
	.	.	.	d	s	s		3:.top/bottom scan
	.	.	.	.	.	d		4:.exit
(Conditional Statement)	.	G	G	G	G	G	3	*(preCondition)*
k := -1 to MAXSCAN	.	t	f	.	.	.		scan permiss'ble
d[k]<>0	.	.	.	.	t	c		dark pt. k <> zero
d[k]<>d[k-2]	.	.	.	.	t	c		dark pt. k <> dark pt (k-2)
(Procedural Statement)	.	S	S	S	S	S	1	*(Procedure)*
SKELETONIZE(PATTERN,j,1,MAXROW,d[k]);	.	.	.	4	3	.		execute the kth scan on the entire pat.
(Imperative Statement)	.	S	S	S	S	S	7	*(Action)*
d[k] := 0;	.	1	.	.	.	.		initialize d to ZERO
i := 0;	.	.	1	.	.	.		initialize the scan number
k := 0;	.	.	2	.	.	.		initialize the pass number
i := i + 1;	.	.	.	1	.	.		increment the pass number by one
j := 0;	.	.	.	2	.	.		set scan type to left/right edgepoints
k := k + 1;	.	.	.	3	2	.		increment the scan number by one
j := 2;	.	.	.	.	1	.		set scan type to top/bottom edgepoints
(Conditional Statement)	.	G	G	G	G	G	.	*(postCondition)*

g) Optimized and documented infoMap

1:.start initialization of dark points
more dark points, CONTINUE AT 1
no more dark points, CONTINUE AT 2

2:.left/right scan
try left/right scan, CONTINUE AT 3

3:.top/bottom scan
try top/bottom scan, CONTINUE AT 2
stop scanning, CONTINUE AT 4

4:.exit

h) Optimized state diagram as Structured English (generated)

1:.start initialization of dark points
more dark points, CONTINUE AT 1
-1 =< MAXSCAN =< k -> d[k] := 0; goto 1

no more dark points, CONTINUE AT 2
-1 > MAXSCAN > k -> i:=0; k:= 0; goto 2

2:.left/right scan
try left/right scan, CONTINUE AT 3
i:= i + 1; j := 0; k := k + 1;
SKELETONIZE(PATTERN ,j, 1, MAXROW, d[k]);

3:.top/bottom scan
try top/bottom scan, CONTINUE AT 2
d[k] <> 0 and d[k] <> d[k-2] -> j:= 2; k:= k+1;
SKELETONIZE(PATTERN, j, 1, MAXROW, d[k]); goto 2

stop scanning, CONTINUE AT 4
d[k]= 0 or d[k]= d[k-2] -> goto 4

4:.exit

i) source code (generated)

Fig 5.3.4.1 Control flow for ONE_PROCESSOR_SPTA procedure

Chapter 6

Required Properties of TAPi Environment

6.1 Productivity of Knowledge Acquisition

The best way to enhance the productivity of creating an algorithm is to avoid building it from scratch. Part of the time spend in writing it should be replaced by time spend to find, understand, modify and compose reusable parts. The TAPi methodology is a step in that direction. The data model is a clear example in which such parts can be reused as algorithms are extended and maintained.

6.2 Quality of TAPi Products:

During modeling, consideration must be given to characteristics which have to do with the quality and performance of algorithms and that are directed toward user satisfaction. To ensure such quality, verification should be carried out at various stages of the modeling process. Results from procedures and processes should be validated.

6.2.1 Verification:

It is fairly simple to verify that the code implements the design at the most detailed level. One can take each design document (flowchart symbol, design language statement, and so on) for a given procedure and find its counterpart in the code. Depending on how detailed the detailed design gets, these specifications are for either individual procedures or collections of procedures identified as a single module in the overall design structure. In either case, it is adviseable to determine if the code will perform the role explicitly assigned to it.

One systematic way to accomplished this is to perform an inspection on a statement-by-statement level. This procedure may be painstaking. If the overall design has poor structure, this may be impractical to accomplish within a reasonable time. With a design that has good structure, one can select a usefully representative set of test cases.

When manually stepping through an algorithm, it is difficult to remember the state of the several data-objects at each point. The TAPi methodology provides an efficient and effective trace facility which is aimed at correcting the problem. The trace records and displays the state of the algorithm at critical times. Fig. 6.2.1.1 demonstrates how the TAPi trace works, using as an example, a Straight Forward search algorithm. In Fig. 6.2.1.2 a search is made for the occurrence of the pattern substring "in" in the string "infoMap". Two paths are executed for this search - path 1 contains three transitions which are fired in sequence and path 2, two transitions. The right hand side of the infoMap records the values of the data-objects before and after transitions are fired. The transitions are sequenced in time. In Fig. 6.2.1.3 the search is made for the occurrence of the pattern substring "Map" in the string "infoMap".

	O	.	.	.	.	.	.	x	*(Path)*
	S	*O*	*O*	*O*	*O*	*O*	*O*	6	*(Transition)*
	.	o	.	.	.	.	.		initialize guess ptr. for pat within the text
	.	.	o	.	.	.	.		location within the text
	.	.	.	o	.	.	.		location outside the text
	.	.	.	.	o	.	.		all pat chars matched
	.	.	.	.	.	o	.		match at current text location
	.	.	.	.	.	.	o		pat elem does not match with text elem
	.	*L*	*L*	*L*	*L*	*L*	*L*	4	*(State)*
	.	s	.	.	.	.	.		1:.Start
	.	d	s	s	.	.	d		2:.evaluating remaining text
	.	.	d	.	s	l	s		3:.matching pat elem with text elem
	.	.	.	d	d	.	.		4:.exit
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	3	*(preCondition)*
i<n-m+1	.	.	t	f	.	.	.		location within text
PAT(k) = TXT(j)	.	.	.	.	t	t	f		pat elem equals text elem
k=m	.	.	.	.	t	f	.		all pat chars compared
(Imperative Statement)	.	*S*	*S*	*S*	*S*	*S*	*S*	8	*(Action)*
i := 0;	.	1	.	.	.	.	.		set guess ptr. to zero
i := i+1	.	.	1	.	.	.	.		increment guess ptr. by 1
j := i;	.	.	2	.	.	.	.		set text ptr to guess ptr. value
k := 1;	.	.	3	.	.	.	.		set pat ptr. to 1
return("pattern not found");	.	.	.	1	.	.	.		pattern not found
return("pattern found at", i)	.	.	.	.	1	.	.		pattern found at guess ptr. value
j := j+1;	.	.	.	.	.	1	.		increment text ptr. by 1
k := k+1;	.	.	.	.	.	2	.		increment pat ptr by 1
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	.	*(postCondition)*

a) Optimized and documented infoMap: shaded area hidden in remaining figures of section 6.2.1

.	.	.	.	.	.	A	A	A	A	A	A	A	2	*(Run)*
.	.	.	.	.	.	1	2	.	.	.	.	.		pattern "in" run
.	.	.	.	.	.	.	.	1	2	3	4	5		pattern "Map" run
.	.	.	.	.	.	O	O	O	O	O	O	O	6	*(Path)*
.	.	.	.	.	.	1	2	3	4	5	6	2		path id.
O	O	O	O	O	O	S	S	S	S	S	S	S	6	*(Transition)*
o	.	.	.	.	.	1	.	1	.	.	.	.		initialize guess ptr. for pat within the text
.	o	.	.	.	.	2	.	2	1	1	.	.		location within the text
.	.	o	.	.	.	.	.	.	.	.	.	.		location outside the text
.	.	.	o	.	.	.	2	.	.	.	.	2		all pat chars matched
.	.	.	.	o	.	3	1	.	.	2	1	1		match at current text location
.	.	.	.	.	o			3	2	.		.		pat elem does not match with text elem
S	S	S	S	S	S	.	.	.	.	.	.	.	8	*(Action)*
1	.	.	.	.	.	.	.	.	.	.	.	.		set guess ptr. to zero
.	1	.	.	.	.	.	.	.	.	.	.	.		increment guess ptr. by 1
.	2	.	.	.	.	.	.	.	.	.	.	.		set text ptr. to guess ptr. value
.	3	.	.	.	.	.	.	.	.	.	.	.		set pat ptr. to 1
.	.	1	.	.	.	.	.	.	.	.	.	.		pattern not found
.	.	.	1	.	.	.	.	.	.	.	.	.		pattern found at guess ptr. value
.	.	.	.	1	.	.	.	.	.	.	.	.		increment text ptr. by 1
.	.	.	.	2	.	.	.	.	.	.	.	.		increment pat ptr. by 1
.	.	.	.	.	.	.	.	.	.	.	.	.	8	*(dataObject - INPUT)*
.	.	.	.	.	.	.	.	.	.	.	.	.		pattern: "In"
.	.	.	.	.	.	.	.	.	.	.	.	.		pattern: "Map"
.	.	.	.	.	.	.	.	.	.	.	.	.		text: "InfoMap"
.	.	.	.	.	.	.	.	.	.	.	.	.		length of pattern substring
.	.	.	.	.	.	.	.	.	.	.	.	.		length of text string
.	.	.	.	.	.	.	.	.	.	.	.	.		guess ptr. for pat within the text
.	.	.	.	.	.	.	.	.	.	.	.	.		text ptr.
.	.	.	.	.	.	.	.	.	.	.	.	.		pat ptr.
.	.	.	.	.	.	.	.	.	.	.	.	.	5	*(dataObject - OUTPUT)*
.	.	.	.	.	.	.	.	.	.	.	.	.		length of pattern substring
.	.	.	.	.	.	.	.	.	.	.	.	.		length of text string
.	.	.	.	.	.	.	.	.	.	.	.	.		guess ptr. for pat within the text
.	.	.	.	.	.	.	.	.	.	.	.	.		text ptr.
.	.	.	.	.	.	.	.	.	.	.	.	.		pat ptr.

b) Paths for patterns execution of the SF algorithm: unshaded area

Fig. 6.2.1.1 Verification of the Straight Forward search algorithm

.	.	.	.	.	.	O	O	2	*{Path}*		
.	.	.	.	.	.	1	2		path id.		
.	.	.	.	.	.	A	A	1	*{Run}*	A	A
.	.	.	.	.	.	v	v		pattern "in" run	v	v
O	O	O	O	O	O	S	S	6	*{Transition}*	T	T
o	.	.	.	.	.	1	.		initialize guess ptr. for pat within the text	1	.
.	o	.	.	.	.	2	.		location within the text	2	.
.	.	o	.	.	.	.	.		location outside the text	.	.
.	.	.	o	.	.	.	2		all pat chars matched	.	5
.	.	.	.	o	.	3	1		match at current text location	3	4
.	.	.	.	.	o	.	.		pat elem does not match with text elem	.	.
S	S	S	S	S	S	.	.	8	*{Action}*	S	S
1	.	.	.	.	.	.	.	.	set guess ptr. to zero	1	.
.	1	.	.	.	.	.	.	.	increment guess ptr. by 1	2	.
.	2	.	.	.	.	.	.	.	set text ptr. to guess ptr. value	3	.
.	3	.	.	.	.	.	.	.	set pat ptr. to 1	4	.
.	.	1	.	.	.	.	.	.	pattern not found	.	.
.	.	.	1	.	.	.	.	.	pattern found at guess ptr. value	.	9
.	.	.	.	1	.	.	.	.	increment text ptr. by 1	5	7
.	.	.	.	2	.	.	.	.	increment pat ptr. by 1	6	8
.	.	.	.	.	.	.	.	5	*{data-object - INPUT}*	V	V
.	.	.	.	.	.	.	.	.	pattern: "in"	.	.
.	.	.	.	.	.	.	.	.	text: "infoMap"	.	.
.	.	.	.	.	.	.	.	.	length of pattern substring	2	2
.	.	.	.	.	.	.	.	.	length of text string	7	7
.	.	.	.	.	.	.	.	.	guess ptr. for pat within the text	.	1
.	.	.	.	.	.	.	.	.	text ptr.	.	2
.	.	.	.	.	.	.	.	.	pat ptr.	.	2
.	.	.	.	.	.	.	.	5	*{data-object - OUTPUT}*	V	V
.	.	.	.	.	.	.	.	.	length of pattern substring	2	2
.	.	.	.	.	.	.	.	.	length of text string	7	7
.	.	.	.	.	.	.	.	.	guess ptr. for pat within the text	1	1
.	.	.	.	.	.	.	.	.	text ptr.	2	3
.	.	.	.	.	.	.	.	.	pat ptr.	2	3

Fig. 6.2.1.2 path verification for SF algorithm: pattern "in" in text "infoMap"

.	.	.	.	.	.	*O*	*O*	*O*	*O*	*O*	5	*(Path)*							
.	.	.	.	.	.	3	4	5	6	2		*path id.*							
.	.	.	.	.	.	*A*	*A*	*A*	*A*	*A*	1	*(Run)*	*A*	*A*	*A*	*A*	*A*	*A*	*A*
.	.	.	.	.	.	v	v	v	v	v		pattern "Map" run	v	v	v	v	v	v	v
O	*O*	*O*	*O*	*O*	*O*	*S*	*S*	*S*	*S*	*S*	6	*(Transition)*	*T*	*T*	*T*	*T*	*T*	*T*	*T*
o		.	.	.	.	1	.	.	.	.		initialize guess ptr. for pat within the text	1	.	.	.	.	.	.
.	o	.	.	.	.	2	1	1	.	.		location within the text	2	4	6	8	10	.	.
.	.	o	.	.	.	.	.	.	.	.		location outside the text	.	.	.	.	.	.	.
.	.	.	o	.	.	.	.	.	.	2		*all pat chars matched*	.	.	.	.	.	.	14
.	.	.	.	o	.	.	.	2	1	1		match at current text location	.	.	.	.	11	12	13
.	.	.	.	.	o	3	2	.	.	.		*pat elem does not match with text elem*	3	5	7	9	.	.	.
S	*S*	*S*	*S*	*S*	*S*	.	.	.	.	.	8	*(Action)*	*S*	*S*	*S*	*S*	*S*	*S*	*S*
1	.	.	.	.	.	.	.	.	.	.		set guess ptr. to zero	1	.	.	.	.	.	.
.	1	.	.	.	.	.	.	.	.	.		increment guess ptr. by 1	2	1	1	1	.	.	.
.	2	.	.	.	.		.	.	.	.		set text ptr. to guess ptr. value	3	2	2	2	.	.	.
.	3	.	.	.	.	.	.	.	.	.		set pat ptr. to 1	4	3	3	3	.	.	.
.	.	1	.	.	.	.	.	.	.	.		pattern not found	.	.	.	.	.	.	.
.	.	.	1	.	.	.	.	.	.	.		*pattern found at guess ptr. value*	.	.	.	.	.	.	.
.	.	.	.	1	.	.	.	.	.	.		increment text ptr. by 1	.	.	.	.	1	.	2
.	.	.	.	2	.	.	.	.	.	.		increment pat ptr. by 1	.	.	.	.	2	1	1
.	.	.	.	.	.	.	.	.	.	.	8	*(data-object - INPUT)*	V	V	V	V	V	V	V
	.	.	.	.	.	.	.	.	.	.		pattern: "Map"	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.		text: "infoMap"	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.		length of pattern substring	3	3	3	3	3	3	3
.	.	.	.	.	.	.	.	.	.	.		*length of text string*	7	7	7	7	7	7	7
.	.	.	.	.	.	.	.	.	.	.		initialize guess ptr. for pat within the text	.	3	2	3	4	5	5
.	.	.	.	.	.	.	.	.	.	.		text ptr.	.	1	2	3	4	5	6
.	.	.	.	.	.	.	.	.	.	.		pat ptr.	.	1	1	1	1	1	2
.	.	.	.	.	.	.	.	.	.	.	5	*(data-object - OUTPUT)*	V	V	V	V	V	V	V
.	.	.	.	.	.	.	.	.	.	.		length of pattern substring	3	3	3	3	3	3	3
.	.	.	.	.	.	.	.	.	.	.		length of text string	7	7	7	7	7	7	7
.	.	.	.	.	.	.	.	.	.	.		*initialize guess ptr. for pat within the text*	1	2	3	4	5	5	5
.	.	.	.	.	.	.	.	.	.	.		text ptr.	1	2	3	4	5	6	7
.	.	.	.	.	.	.	.	.	.	.		pat ptr.	1	1	1	1	1	2	3

Fig. 6.2.1.3 Path verification for SF algorithm: pattern "Map" in text "infoMap"

6.2.2 Validation:

Validation may be defined as checks which are performed at each stage or phase of the development process in order to determine the satisfaction of the requirements specification. The auditor needs reassurance that the algorithm does what it is supposed to do.

One would look for inconsistencies within the specifications and deviations from any standards of documentation. Design documentation at all levels would have to be closely examined. Since TAPi provides for documenting and coding within its models, the problem of inconsistencies is thus eliminated. For instance, in a conventional environment, modification(s) carried out at the code level may not necessarily update the counterpart(s) at the design level. This leads to inconsistencies. However, in TAPi the changes takes place at the model level and thus, no inconsistencies should arise.

Very often it is not easy to communicate with others through code or pseudocode. The TAPi methodology provides the means for the user to record the code and give meaning to it. This allows the user to validate his or her understanding of the code with the appropriate authority.

6.3 Interface

This section considers and proposes required properties for two interfaces. They are Graphic User Interface and Dynamic infoMap (Animation). These are dicussed below.

6.3.1 Graphic User Interface:

In the development of the understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction. - C.A.R. Hoare, Notes on Data Structuring, 1972 [51]. For example, a functional abstraction may be specified by an input-output relation. The user is aware only of the input-output specification and not of the way the function is implemented. The specification constitutes the interface to the user. The implementation is hidden from the user.

TAPi models are in the form of infoMaps. Each infoMap is constructed using a spreadsheet or at least a block oriented editor. Such interfaces would allow the user to manipulate the spreadsheet-based infoMap efficiently and effectively.

6.3.2 Dynamic infoMap (Animation):

The provision for feedback during the modeling process would be of tremendous benefit to a developer or user. Therefore, actions should be recorded and displayed in windows on the screen or provision be made for the option of instant replay(s) of the actions. The purpose is to allow the user to meticulously examine the actions so as to eliminate unnecessary and redundant ones in order to improve subsequent modeling attempts.

During the processing of the optimized infoMap, the various paths taken should be displayed. The sequencing of the transitions as they are fired should also be displayed together with their guards, actions and so on. Some transitions may or may not be guarded. Such displays can be shown in windows and serve the purpose of motivating a developer or user.

Chapter 7

Conclusions

It is evident from the infoMap illustrations in this thesis that the proposed methodology (TAPi) is an excellent methodology for the transformation of algorithms and programs. It needs to be emphasized that the steps of the methodology must be followed in order to guarantee successful transformation. For instance, if a data model is not built or if the attributes or data- objects built are not given proper meaning then at a later stage it would become difficult to attach meaning to transitions, states, conditions, actions and so on.

It is also apparent from the foregoing analysis, that many algorithms are not built from scratch but are only modified versions of what already exists. They inherit attributes from their predecessors. Do not 're-invent the wheel' if parts can be reused.

It was discovered that the number of lines in an algorithm is not the only metric to determine its complexity. By transforming the algorithm into infoMaps, other metrics such as transitions and states could give a better measure of their complexity.

Another striking discovery made was that conventional programming environments do not allow one to document algorithm(s) properly. The TAPi methodology, right from the beginning is providing the means for documenting algorithms - the result being well documented algorithms that can be easily understood.

The research was limited to only the transformation of algorithms. However, such a methodology may be applied to systems comprising many modules and procedures. The modules can be treated as chunk of knowledge or algorithms and thus, the steps outlined and discussed in this thesis can be followed.

Future research in the area of infoMaps should focus on the following:

i) formalization of the transformation process described in chapter 3 and illustrated in chapter 5. The use of infoSchema as a structure and generated infoFrames as a repository are aimed at forcing the user or developer to fill in textual descriptions communicating his or her understanding of the problem. The process could be formalized and is suggested as a topic for further research.

ii) interface issues (see section 6.3) such as:

a) graphical windows displaying transitions as they are fired and the statements executed.

b) automating infoMaps structures i.e. these should be generated from parameters supplied by the developer or user.

REFERENCES

[1] Ackerman, A.F., Buchwald, L.S., Lewski, F.H. "Software Inspections: An Effective Verification Process", IEEE Software, pp. 31-36, May 1989.

[2] ACM Forum, Comm.ACM 30, 5 May 1987, 351-5.

[3] Basili, V.R., Mills, H.D. "Understanding and Documenting Programs", IEEE Trans. on SE, vol. 8, no. 3, pp. 270-283, May 1982.

[4] Beffert, H. and Shinghal, R. "Skeletonizing binary patterns on the homogeneous multiprocessor, International Journal of Pattern Recognition and Artificial Intelligence Vol. 3 no. 2 (1989), pp 207-216

[5] Biggerstaff, T. J. and Perlis, A. J. "Software Reusability, Volume 1 - Concepts and Models", ACM Press, 1989

[6] Borgida, A., Greenspan, S., Mylopoulos, J. "Knowledge Representation as the Basis for Requirements Specifications", Computer, pp. 82-91, April 1985.

[7] Buck, R.D., Dobbins, J.H. "Application of Software Inspection Methodology in Design and Code", Software Validation, Elsevier Science Publishers B.V., New York, 1984, pp. 41-56.

[8] Buckley, F.J. "Some Standards for Software Maintenance", IEEE Computer, pp. 69-70, November 1989.

[9] Buckley, F.J. "Do standards cause software productivity problems?", Computer, vol. 24, no. 1, pp. 97-98, January 1991.

[10] Buckley, F.J., Poston, R. "Software Quality Assurance", IEEE Trans. on SE, vol. 10, no. 1, pp. 36-41, January 1984.

[11] Carrico, M.A., Girard, J.E. and Jones, J.P. "Building Knowledge Systems Developing and Managing Ruled-Based Applications", McGraw-Hill Book Company, 1989

[12] Chandrasekharan, M., Dasarathy, B., Kishimoto, Z. "Requirements-Based Testing of Real-Time Systems: Modeling for Testability", Computer, pp. 71-80, April 1985.

[13] Coad, P. and Yourdon, E. "Object-oriented Analysis", Prentice-Hall, Inc., 1990

[14] Cooper, D.C. "Bohm and Jacopini's Reduction of Flowcharts", Communications of the ACM, August 1967

[15] Craigen, D. "Strengths and Weaknesses of Program Verification Systems", 1st European Software Engineering Conference, pp. 397-404, 1987.

[16] Crosby, M.E., Stelovsky, J. "How Do We Read Algorithms? A Case Study", Computer, vol. 23, no. 1, pp. 24-35, January 1990.

[17] Darst, D. "Balancing Productivity and Quality", Datamation, vol. 36, no. 18, pp. 117-119, September 15, 1990.

[18] Dijkstra, E.W., "Guarded commands, nondeterminancy and the formal derivation of programs", Comm. ACM 18 (August 1975), 453-457.

[19] Eisenstadt, Domingue, Rajan, Motta, "Visual Knowledge Engineering", IEEE Transactions on Software Engineering Vol.16 No.10 October 1990

[20] Elliott, I. B. "The EPN and ESN notations." SIGPLAN Notices 19, 7 Jul 84, 9-17.

[21] Fagan, M.E. "Design and code inspections to reduce errors in program development", IBM Systems Journal, vol. 15, no. 3, pp. 182-211, 1976.

[22] Freeman, P. "Establishing a System for Developing Software", Software Engineering, vol. 1, no. 2, pp. 14-22, July/August 1990.

[23] Gane, C. "Rapid System Development Using Structured Techniques and relational Technology ", Prentice Hall, 1989

[24] Gane, C. and Sarson, T. "Structured Systems Analysis: Tools and Techniques", Prentice-Hall, Inc., 1979

[25] Grogono, P., Jaworski, W.M.,Software Development as Knowledge Acquisition using jMaps, Canadian Conference on Electrical and Computer Engineering, Montreal, September 17-20, 1989.

[26] Humphrey, W.S., M.I. Kellner, "Software Process Modeling: Principles of Entity Process Models", Proc. 11th Conf. Software Eng., Pittsburgh, Pa., New-York: IEEE Press, pp. 331-342, 1989.

[27] "IEEE Standard for Software Verification and Validation Plans", ANSI/IEEE Std 1012-1986, Software Engineering Standard, IEEE, 1987.

[28] Jaworski, W.M., Virard M., "Converting a Software Company to a New Technology", Proceedings of the 1986 Canadian Conference on Industrial Computers, 1986, Montreal pp 12-1..12-7.

[29] Jaworski, W.M., Farell, E., "Development of Spectrum Management Systems: Application of J-map Notational Technology", Montech'87 Proceedings, Conference on Communications, November 9-11, 1987, Montreal, pp. 181-185.

[30] Jaworski, W.M. "Systems Analysis and Design in the Classroom: infoMAPs Factory", Modeling and Simulation Conference, Pittsburgh, Pa., May 3-4, 1990.

[31] Jaworski, W.M. and Deslauriers, B "Software: Process, Maintenance, Products, infoMaps", Canadian Conference on Electrical and Computer Engineering, Ottawa, Ont., Sept. 4-6, 1990.

[32] Jonsson, D. "Pancode and Boxcharts: structured programming revisited", SIGPLAN Notices 22, 8 August 1987, 89-98.

[33] Jonsson, D. "Pancode assessed." SIGPLAN Notices 24, 12 December 1989, 17-20.

[34] Kovats, T. "Comments on innovative control constructs in Pancode and EPN". SIGPLAN Notices 23, 12 December 1988, 151-157.

[35] Kovats, T.A. "A conservative Alternative to Pancode", SIGPLAN Notices 25, 11 November 1990.

[36] Lichtman, Z.L. "Generation and Consistency Checking of Design and Program Structures", IEEE Trans. on SE, vol. 12, no. 1, pp.172-181, January 1986.

[37] Matthews, R.W and McGee, W.C. "Data modeling for software development", IBM Systems Journal, vol. 29, no. 2, pp. 228-235, 1990.

[38] Mays, R.G., Orzech, L.S., Ciarfella, W.A., Phillips, R.W. "PDM: A requirements methodology for software system enhancements", IBM System Journal, vol. 24, no. 2, pp. 134-149, 1985.

[39] Narula, S.C. and Ho, C.A. "Degree-Constrained Minimum Spanning Tree", Comput. & Ops. Res. Vol. 7, pp 239-249

[40] Osterweil, L. "Integrating the Testing, Analysis and Debugging of

Programs", Software Validation, Elsevier Science Publishers B.V., New York, 1984, pp. 73-102.

[41] Reynolds, R.G., Maletic, J.I. and Porvin, S.E. "PM: A System to Support the Automatic Acquisition of Programming Knowledge", IEEE Transactions of Knowledge and Data Engineering, Vol.2. No.3 September 1990

[42] Roman, G.C. "A Taxonomy of Current Issues in Requirements Engineering", Computer, pp. 14-21, April 1985.

[43] Rubin, F. "GOTO considered harmful" considered harmful. Comm. ACM 30, 3 March 1987, 195-196.

[44] Rumbaugh, J. et al "Object-oriented modeling and design", Prentice-Hall, Inc. 1991

[45] Russell, G.W. "Experience with Inspection in Ultralarge- Scale Developments", IEEE Software, pp. 25-31, January 1991.

[46] Rzepka, W., Ohno, Y. "Requirements Engineering Environments: Software Tools for Modeling User Needs", Computer, pp. 9-12, April 1985.

[47] Silverman, B.G. "Software Cost and Productivity Improvements: An Analogical View", Computer, vol. 18, no.5, pp. 86-96, May 1985.

[48] Smit, G.V. "A comparison of three string matching algorithms", Soft-Prac. and Exp, Vol.12, 57-66 (1982)

[49] Sunday, D.M. "A very fast Substring Search Algorithm", Communications of the ACM, Vol 33 No.8 August 1990

[50] Webster, D.E., "Mapping the Design Representation Terrain", Technical Report STP-093-87, MCC, Software Technology Program, July 1987.

[51] Weinberg, G.M., Freedman, D.P. "Reviews, Walkthroughs, and Inspections", IEEE Trans. on SE, vol. 10, no. 1, pp. 68-72, January 1984.

[52] Yau, S.S., Collofello, J.S "Design Stability Measures for Software Maintenance", IEEE Trans. on SE, vol. 11, no. 9, pp. 849-856, September 1985.

Appendix I

Syntax of infoMaps notation

This appendix contains one infoMap which gives the syntax of the infoMap notation. For instance, Set_Role "A" which means Associative structures, can have Set_member_Roles "v", "o,O", "m" and "m;n". Considering another example, Set_Role "O" which means Dominant set can have Set_member_Role "o" which means dominant set member.

	/	/	/	/	/	/	/	/	/	/	/	*{Set_Role}*
	p											*A::= Associative structure (of structures)*
		p										*H::= Hierarchical structure*
			p									*O::= Dominant set*
				p								*F::= Flow definition*
					p							*I::= Inheritance structure*
						p						*L::= directed graph with cycles and forks*
							p					*G::= Guard or post-Guard*
								p				*S::= Sequence*
									p			*R::= Run structure*
										p		*V::= Value or instance*
	/	/	/	/	/	/	/	/	/	/	/	*{Set_member_Role}*
	c											*v::= column marker.*
	c											*o,O::= relationship cardinality definition: "one to .. ", "One to ..."*
	c											*m::= relationship cardinality definition: "many to ..."*
	c											*m;n::= range definition: "at least m; at most n"*
		c										*v::= cell marker.*
		c										*h::= root of a hierarchical structure*
		c										*1 .. n::= element identification*
			c									*o::= dominant set member*
				c								*i,o,b::= flow direction: "in","out", "in/out"*
					c							*p,c,e::= inheritance flow: "parent","child","extends"*
						c						*s,d,l,a,e::= "source","destination","loop","assertion","exception"*
							c					*t,f,c,T,F::="true","false","complementary","asserted t","asserted f"*
	c							c				*1..n::= "place in sequence"*
									c			*el,dl::="enabled","disabled"*
										c		*integer, real, char*
												Copyright © W.M. Jaworski 1990

Appendix II

Program Normalization and Optimization

This appendix consists of two figures. The first, Fig. 4.3.2, shows the transformation of the All-zero row program (in Pascal and Pancode) into infoMaps. The second, Fig. 4.3.3, shows the transformation of the program (in EPN style) into infoMaps. Statistics from these infoMaps are summarized in Chapter 4, section 4.4.

```
1   i := 1;
2   repeat
      j := 1;
3   while(j <= n
      and (x[i, j] = 0) do
        j := j + 1;
    i := i + 1;
4 until (i > n) or (j > n);
5. if j > n
   then writeln
   ('The .. row is', i - 1 );
6
```

(a) Pascal-type code

```
L  L  L  L  L  L  L  L   (Node)
s                           1
d  s           d            2
   d  l  s                  3
         d  s  s            4
               d  s  s      5
                  d  d      6
G  G  G  G  G  G  G  G   (preCondition)
      t  f                  j<=n and x[i,j]=0
            f  t            i>n or j>n
                  t  f      j>n
S  S  S  S  S  S  S  S   (Action Statement)
1                           i =1
   1                        j =1
      1                     j =j+1,
         1        1         i =i+1,
                            Print ('The   row is', i-1)
```

(b) Pascal code rewritten as infoMap

```
                      i:=1
do checkrow:          if i<=N
                      j:=1
do checknumber:  if j<=N
                           if x[i,j]=o
                            j:=j+1
                 repeat checknumber
                      i:=i+1
                 repeat checkrow
                  print('Row', i)
```

(c) Pancode

```
L  L  L  L  L  L   (Node)
s
d  s  s        d
      d  s  l  s
   d  .  d
G  G  G  G  G  G   (preCondition)
   f  t               i <= N
         f  t  t      j <= N
            f  t      X[i,j]<>0
S  S  S  S  S  S   (Action Statement)
1                     i = 1
      1               j =1
            1         j = j+1
               1      i := i+1
         1  .         print('Row, i)
```

(d) Pancode rewritten as infoMap

Fig. 4.3.2 All-zero row Program

```
if true then
  for i:= 1 to N
    for j:= 1
      fail on j > N;
        exit on X[i,j]<>0
    next j
  next i
else
  print('Row',i)
endif
```

(a) EPN code

```
L L L L L L [Node]
s
d s s       d
    d s d
      s s s
  d   e
G G G G G G [preCondition]
s             true
  1 1         i<=N
        1 1   X[i,j]<>0
S S S S S S [Action Statement]
1             i = 1
    1         j =1
        1     j = j+1
          1   i = i+1
      1       print(Row,i)
G G G G G G [postCondition]
      1       j <= N
```

(b) EPN code rewritten as infoMap

```
for i   1 to N
  for j   1
    if j>N then
        print('Row',i);
      exit 2
    endif;
    exit on X[i,j]<>0
  next j
next i
```

(c) shorter EPN code

```
L L L L L L  [Node]
s
d s s     d
    d s 1 s
  d   d
G G G G G G  [preCondition]
  1 1          i<= N
      1 1 1    j<= N
        1 1    X[i,j]<>0
S S S S S S  [Action Statement]
1              i = 1
    1          j =1
        1      j = j+1
          1    i = i+1
      1        print(Row,i)
```

(d) shorter EPN code as infoMap

```
O O O O [Branch]
o          first element in first row
  o        try next element in the row
    o      try next row unless no more
      o    all zero row
I I I I [Node]
s          start
d 1 1 s    process X[i,j]
    e d    end
G G G G [preCondition]
  1   1    j<=n  % "column number <= n"
  1 1      x[i,j]=0 % "current matrix element = 0" }
S S S S [Action Statement]
1          i=1,  % "first row"
1   1      j=1,  % "first element in row"
  1        j=j+1,  % "next element in row
    1      i=i+1,  % "next row"
      1    Print ("The first all zero row is", i )
G G G G [postCondition]
    1      i>n  % "row number > n"
```

(e) optimized and documented infoMap

Fig. 4.3.3 All-zero row Program in EPN Style

Appendix III

String Search algorithms

This appendix contains the transformation of four string search algorithms and procedures. The procedure(s) corresponding to an algorithm has to be executed before the execution of the algorithm. The statistics obtained from these transformed algorithms and procedures are summarized in chapter 5, section 5.1.4.

```
1.      j := 1; i := FAIL(1) := 0;
2.      while(j < m) do
        /* FAIL(1) ... FAIL(j) is known - compute FAIL(j+1) */
3.          while(i > 0 and PAT(j) <> PAT(i)) do
4.              i := FAIL(i); end;
5.          i := i+1; j := j+1;
6.          if(PAT(j) = PAT(i)) then
7.              FAIL(j) := FAIL(i)
            else
8.              FAIL(j) := i;
        end;
```

a) source code: original

```
1:      j := 1; i := FAIL(1) := 0;
2:      while(j < m) do
        /* FAIL(1) ... FAIL(j) is known - compute FAIL(j+1) */
3:          while(i > 0 and PAT(j) <> PAT(i)) do
                i := FAIL(i); end;
            i := i+1; j := j+1;
4:          if(PAT(j) = PAT(i)) then
                FAIL(j) := FAIL(i)
            else
                FAIL(j) := i;
        end;
5:
```

b) edited source code: line nos removed, states 1 . . 5 identified

Note: Nos. 1 .. 8 in a) are line nos. of original algorithm and should not be confused with states 1 . . 5 in b)

	O	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	7	*(transition)*
	.	o	.	.	.	.	.	.		
	.	.	o	.	.	.	.	.		
	.	.	.	o	.	.	.	.		
	.	.	.	.	o	.	.	.		
	.	.	.	.	.	o	.	.		
	.	.	.	.	.	.	o	.		
	.	.	.	.	.	.	.	o		
	.	L	L	L	L	L	L	L	5	*(State)*
	.	s	.	.	.	.	.	.		1:.
	.	d	s	s	.	.	d	d		2:.
	.	.	d	.	i	s	.	.		3:.
	.	.	.	.	.	d	s	s		4:.
	.	.	.	d	.	.	.	.		5:.
(Conditional Statement)	.	G	G	G	G	G	G	G	4	*(preCondition)*
j < m	.	.	t	f	.	.	.	.		more elements left
i > 0	.	.	.	.	t	c	.	.		left pointer greater than zero
PAT(j) <> PAT(i)	.	.	.	.	t	c	.	.		element j not equal to element i
PAT(j) = PAT(i)	.	.	.	.	.	.	t	f		element j equals element i
(Imperative Statement)	.	S	S	S	S	S	S	S	8	*(Action)*
j := 1;	.	1	.	.	.	.	.	.		set right pointer to 1
i := 0;	.	2	.	.	.	.	.	.		set left pointer to zero
FAIL(1) := 0;	.	3	.	.	.	.	.	.		set first element of Fail to zero
i := FAIL(i);	.	.	.	.	1	.	.	.		calculate new left pointer
i := i+1;	.	.	.	.	.	1	.	.		increment left pointer by 1
j := j+1;	.	.	.	.	.	2	.	.		increment right pointer by 1
FAIL(j) := FAIL(i)	.	.	.	.	.	.	1	.		set j elem. of Fail to i elem. of Fail
FAIL(j) := i;	.	.	.	.	.	.	.	1		set j elem. of Fail to left pointer
(Conditional Statement)	.	G	G	G	G	G	G	G	.	*(postCondition)*

c) Normalized infoMap: shaded areas to be filled in

	O	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	7	*(transition)*
	.	o	.	.	.	.	.	.		initialize pointers
	.	.	o	.	.	.	.	.		more elements left
	.	.	.	o	.	.	.	.		no more elements left
	.	.	.	.	o	.	.	.		no update to pointers
	.	.	.	.	.	o	.	.		update pointers
	.	.	.	.	.	.	o	.		element j equals element i
	.	.	.	.	.	.	.	o		element j not equal to element i
	.	L	L	L	L	L	L	L	5	*(State)*
	.	s	.	.	.	.	.	.		1:.start initialization of pointers
	.	d	s	s	.	.	d	d		2:.evaluation of remaining elements
	.	.	d	.	i	s	.	.		3:.updating pointers
	.	.	.	.	.	d	s	s		4:.comparison of elements
	.	.	.	d	.	.	.	.		5:.exit
(Conditional Statement)	.	G	G	G	G	G	G	G	4	*(preCondition)*
j < m	.	.	t	f	.	.	.	.		more elements left
i > 0	.	.	.	.	t	c	.	.		left pointer greater than zero
PAT(j) <> PAT(i)	.	.	.	.	t	c	.	.		element j not equal to element i
PAT(j) = PAT(i)	.	.	.	.	.	.	t	f		element j equals element i
(Imperative Statement)	.	S	S	S	S	S	S	S	8	*(Action)*
j := 1;	.	1	.	.	.	.	.	.		set right pointer to 1
i := 0;	.	2	.	.	.	.	.	.		set left pointer to zero
FAIL(1) := 0;	.	3	.	.	.	.	.	.		set first element of Fail to zero
i := FAIL(i);	.	.	.	.	1	.	.	.		calculate new left pointer
i := i+1;	.	.	.	.	.	1	.	.		increment left pointer by 1
j := j+1;	.	.	.	.	.	2	.	.		increment right pointer by 1
FAIL(j) := FAIL(i)	.	.	.	.	.	.	1	.		set j elem. of Fail to i elem. of Fail
FAIL(j) := i;	.	.	.	.	.	.	.	1		set j elem. of Fail to left pointer
(Conditional Statement)	.	G	G	G	G	G	G	G	.	*(postCondition)*

d) Normalized infoMap: states and transitions added (shaded areas)

	O	.	.	.	.	.	.	.	x	*(Path)*
	S	*O*	*O*	*O*	*O*	*O*	*O*	*O*	7	*(transition)*
	.	o	.	.	.	.	.	.		initialize pointers
	.	.	o	.	.	.	.	.		more elements left
	.	.	.	o	.	.	.	.		no more elements left
	.	.	.	.	o	.	.	.		no update to pointers
	.	.	.	.	.	o	.	.		update pointers
	.	.	.	.	.	.	o	.		element j equals element i
	.	.	.	.	.	.	.	o		element j not equal to element i
	.	*L*	*L*	*L*	*L*	*L*	*L*	*L*	5	*(State)*
	.	s	.	.	.	.	.	.		1:.start initialization of pointers
	.	d	s	s	.	.	d	d		2:.evaluation of remaining elements
	.	.	d	.	l	s	.	.		3:.updating pointers
	.	.	.	.	.	d	s	s		4:.comparison of elements
	.	.	.	d	.	.	.	.		5:.exit
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	4	*(preCondition)*
j < m	.	.	t	1	.	.	.	.		more elements left
i > 0	.	.	.	.	l	c	.	.		left pointer greater than zero
PAT(j) <> PAT(i)	.	.	.	.	l	c	.	.		element j not equal to element i
PAT(j) = PAT(i)	.	.	.	.	.	.	t	f		element j equals element i
(Imperative Statement)	.	*S*	*S*	*S*	*S*	*S*	*S*	*S*	8	*(Action)*
j := 1;	.	1	.	.	.	.	.	.		set right pointer to 1
i := 0;	.	2	.	.	.	.	.	.		set left pointer to zero
FAIL(1) := 0;	.	3	.	.	.	.	.	.		set first element of Fail to zero
i := FAIL(i);	.	.	.	.	1	.	.	.		calculate new left pointer
i := i+1;	.	.	.	.	.	1	.	.		increment left pointer by 1
j := j+1;	.	.	.	.	.	2	.	.		increment right pointer by 1
FAIL(j) := FAIL(i)	.	.	.	.	.	.	1	.		set j elem. of Fail to i elem. of Fail
FAIL(j) := i;	.	.	.	.	.	.	.	1		set j elem. of Fail to left pointer
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	.	*(postCondition)*

e) Normalized infoMap: state diagram (unshaded area)

	O	.	.	.	.	.	.	.	x	*(Path)*
	S	*O*	*O*	*O*	*O*	*O*	*O*	*O*	7	*(transition)*
	.	o	.	.	.	.	.	.		initialize pointers
	.	.	o	.	.	.	.	.		more elements left
	.	.	.	o	.	.	.	.		no more elements left
	.	.	.	.	o	.	.	.		no update to pointers
	.	.	.	.	.	o	.	.		update pointers
	.	.	.	.	.	.	o	.		element j equals element i
	.	.	.	.	.	.	.	o		element j not equal to element i
	.	*L*	*L*	*L*	*L*	*L*	*L*	*L*	5	*(State)*
	.	s	.	.	.	.	.	.		1:.start initializaiton of pointers
	.	d	s	s	.	.	d	d		2:.evaluation of remaining elements
	.	.	d	.	l	s	.	.		3:.updating pointers
	.	.	.	.	.	d	s	s		4:.comparison of elements
	.	.	.	d	.	.	.	.		5:.exit
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	4	*(preCondition)*
j < m	.	.	t	1	.	.	.	.		more elements left
i > 0	.	.	.	.	l	c	.	.		left pointer greater than zero
PAT(j) <> PAT(i)	.	.	.	.	l	c	.	.		element j not equal to element i
PAT(j) = PAT(i)	.	.	.	.	.	.	t	f		element j equals element i
(Imperative Statement)	.	*S*	*S*	*S*	*S*	*S*	*S*	*S*	8	*(Action)*
j := 1;	.	1	.	.	.	.	.	.		set right pointer to 1
i := 0;	.	2	.	.	.	.	.	.		set left pointer to zero
FAIL(1) := 0;	.	3	.	.	.	.	.	.		set first element of Fail to zero
i := FAIL(i);	.	.	.	.	1	.	.	.		calculate new left pointer
i := i+1;	.	.	.	.	.	1	.	.		increment left pointer by 1
j := j+1;	.	.	.	.	.	2	.	.		increment right pointer by 1
FAIL(j) := FAIL(i)	.	.	.	.	.	.	1	.		set j elem. of Fail to i elem of Fail
FAIL(j) := i;	.	.	.	.	.	.	.	1		set j elem. of Fail to left pointer
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	.	*(postCondition)*

f) Normalized infoMap to be optimized: shaded area

	o	.	.	.	.	.	.	x	*{Path}*
	s	o	o	o	o	o	o	6	*{transition}*
	.	o	.	.	.	.	.		initialize pointers
	.	.	o	.	.	.	.		no more elements left
	.	.	.	o	.	.	.		adjacent characters are different
	.	.	.	.	o	.	.		update pointers
	.	.	.	.	.	o	.		element j equals element i
	.	.	.	.	.	.	o		element j not equal to element i
	.	L	L	L	L	L	L	4	*{State}*
	.	s	.	.	.	.	.		1:.start initialization of pointers
	.	d	s	s	s	d	d		2:.evaluation of remaining elements
	.	.	.	d	d	s	s		3:.updating Fail array
	.	.	d	.	.	.	.		4:.exit
{Conditional Statement}	.	G	G	G	G	G	G	3	*{preCondition}*
j < m	.	.	f	t	t	.	.		more elements left
i > 0	.	.	.	t	c	.	.		left pointer greater than zero
PAT(j) = PAT(i)	.	.	.	f	c	t	f		element j equals element i
{Imperative Statement}	.	S	S	S	S	S	S	8	*{Action}*
j := 1;	.	1	.	.	.	.	.		set right pointer to 1
i := 0;	.	2	.	.	.	.	.		set left pointer to zero
FAIL(1) := 0;	.	3	.	.	.	.	.		set first element of Fail to zero
i := FAIL(i);	.	.	.	1	.	.	.		calculate new left pointer
i := i+1;	.	.	.	.	1	.	.		increment left pointer by 1
j := j+1;	.	.	.	.	2	.	.		increment right pointer by 1
FAIL(j) := FAIL(i)	.	.	.	.	.	1	.		set j elem. of Fail to i elem. of Fail
FAIL(j) := i;	.	.	.	.	.	.	1		set j elem. of Fail to left pointer
{Conditional Statement}	.	G	G	G	G	G	G	.	*{postCondition}*

g) Optimized and documented infoMap

1:. start initialization of pointers
initialize pointers, CONTINUE AT 2

2:. evaluation of Remaining elememts
no more elements left, CONTINUE AT 4
adjacent characters are different, CONTINUE AT 3
update pointers, CONTINUE AT 3

3:. updating Fail array
element j equals element i, CONTINUE AT 2
element j not equal to element i, CONTINUE AT 2

4:. Exit

h) Optimized State diagram as Structured English (generated)

1:. start initialization of pointers

initialize pointers, CONTINUE AT 2

j:= 1; i:= 0; FAIL(1):= 0; goto 2

2:. evaluation of Remaining elememts

no more elements left, CONTINUE AT 4

j>= m -> goto 4

adjacent characters are different, CONTINUE AT 3

< m and i>0 and PAT(j) <> PAT(i) -> i:= FAIL(i); goto 3

update pointers, CONTINUE AT 3

j< m and (i<0 or PAT(j) = PAT(i)) -> i:= i+1; j:= j+1; goto 3

3:. updating Fail array

element j equals element i, CONTINUE AT 2

PAT(j) = PAT(i) -> FAIL(j) := FAIL(i); goto 2

element j not equal to element i, CONTINUE AT 2

PAT(j) <> PAT(i) -> FAIL(j) := i; goto 2

4:. Exit

i) source code (generated)

Fig. 5.1.12 Control flow of Computing table for KMP pattern matching

```
1.    j := k := 1;
2.    while(k <= n) do
3.          while(j > 0 and TXT(k) <> PAT(j)) do
4.                j := FAIL(j);
            end;
5.          if(j = m) then
6.                return("pattern found at", k - m)
            else do
7.                k := k+1;
8.                j := j+1;
            end;
      end;
9.    return("pattern not found");
```

a) source code: original

```
1:    j := k := 1;
2:    while(k <= n) do
3:          while(j > 0 and TXT(k) <> PAT(j)) do
                  j := FAIL(j);
            end;
4:          if(j = m) then
                  return("pattern found at", k - m)
            else do
                  k := k+1;
                  j := j+1;
            end;
      end;
5:    return("pattern not found");
```

b) edited source code: line nos removed, states 1 .. 5 identified

Note: Nos. 1 .. 9 in a) are line numbers of original algorithm and should not be confused with states 1 . . 5 in b).

```
                                o . . . . . . . . x {Path}
                                s o o o o o o o 7 {Transition}
                                . o . . . . . .
                                . . o . . . . .
                                . . . o . . . .
                                . . . . o . . .
                                . . . . . o . .
                                . . . . . . o .
                                . . . . . . . o
                                . L L L L L L L 5 {State}
                                . s . . . . . .     1:
                                . d s s . . . d     2:
                                . . d . l s . .     3:
                                . . . . . d s s     4:
                                . . . d . . d .     5:
{Conditional Statement}         . G G G G G G G 4 {preCondition}
  k <= n                        . . t f . . . .     enough text left
  j>0                           . . . . f c . .     pat pointer > zero
  TXT(k) <> PAT(j)              . . . . f c . .     text elem. not equal to pat elem.
  j = m                         . . . . . . t f     all pat chars compared
{Procedural Statement}          . S S S S S S S 1 {Procedure}
                                . 1 . . . . . .     compute table for KMP pat matching
{Imperative Statement}          . S S S S S S S 7 {Action}
  j := 1;                       . 2 . . . . . .     pat pointer set to 1
  k := 1;                       . 3 . . . . . .     text pointer set to 1
  return("pattern not found");  . . . 1 . . . .     return failure message
  j := FAIL(j);                 . . . . 1 . . .     obtain value of shift
  return("pattern found at", k-m);
                                . . . . . . 1 .     return success message
  k := k+1;                     . . . . . . . 1     increment text pointer
  j := j+1;                     . . . . . . . 2     increment pat pointer
{Conditional Statement}         . G G G G G G G . {postCondition}
```

c) Normalized infoMap: shaded areas to be filled in

Note: "c and c" means all other conditions besides "f and f"

```
                                o . . . . . . . . x {Path}
                                s o o o o o o o 7 {Transition}
                                                    compute table and initialize pointers
                                . o . . . . . .     enough text left
                                . . o . . . . .     not enough text left
                                . . . o . . . .     pat elem not equal to text elem
                                . . . . o . . .     pat elem equals text elem
                                . . . . . o . .     all pat chars matched
                                . . . . . . o .     not all pat chars compared
                                . . . . . . . o
                                . L L L L L L L 5 {State}
                                                    1:.Start
                                . s . . . . . .     2:.Evaluating remaining text
                                . d s s . . . d     3:.Matching text elem with pat elem
                                . . d . l s . .     4:.Testing for pattern occurrence
                                . . . . . d s s     5:.Exit
                                . . . d . . d .
{Conditional Statement}         . G G G G G G G 4 {preCondition}
  k <= n                        . . t f . . . .     enough text left
  j>0                           . . . . f c . .     pat pointer > zero
  TXT(k) <> PAT(j)              . . . . f c . .     text elem not equal to pat elem
  j = m                         . . . . . . t f     all pat chars compared
{Procedural Statement}          . S S S S S S S 1 {Procedure}
                                . 1 . . . . . .     compute table for KMP pat matching
{Imperative Statement}          . S S S S S S S 7 {Action}
  j := 1;                       . 2 . . . . . .     pat pointer set to 1
  k := 1;                       . 3 . . . . . .     text pointer set to 1
  return("pattern not found");  . . . 1 . . . .     return failure message
  j := FAIL(j);                 . . . . 1 . . .     obtain value of shift
  return("pattern found at", k-m);
                                . . . . . . 1 .     return success message
  k := k+1;                     . . . . . . . 1     increment text pointer
  j := j+1;                     . . . . . . . 2     increment pat pointer
{Conditional Statement}         . G G G G G G G . {postCondition}
```

d) Normalized infoMap: states and transitions added (shaded area)

	O	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	7	*(Transition)*
	.	o	.	.	.	.	.	.		compute table and initialize pointers
	.	.	o	.	.	.	.	.		enough text left
	.	.	.	o	.	.	.	.		not enough text left
	.	.	.	.	o	.	.	.		text elem not equal to pat elem
	.	.	.	.	.	o	.	.		pat elem equals text elem
	.	.	.	.	.	.	o	.		all pat chars matched
	.	.	.	.	.	.	.	o		not all pat chars compared
	.	L	L	L	L	L	L	L	5	*(State)*
	.	s	.	.	.	.	.	.		1:.Start
	.	d	s	s	.	.	.	d		2:.Evaluating remaining text
	.	.	d	.	l	s	.	.		3:.Matching text elem with pat elem
	.	.	.	.	.	d	s	s		4:.Testing for pattern occurrence
	.	.	.	d	.	.	d	.		5:.Exit
(Conditional Statement)	.	G	G	G	G	G	G	G	4	*(preCondition)*
k <= n	.	.	t	t	.	.	.	.		enough text left
j>0	.	.	.	.	t	c	.	.		pat pointer > zero
TXT(k) <> PAT(j)	.	.	.	.	t	c	.	.		text elem does not match with pat elem
j = m	.	.	.	.	.	.	t	t		all pat chars compared
(Procedural Statement)	.	S	S	S	S	S	S	S	1	*(Procedure)*
	.	1	.	.	.	.	.	.		compute table for KMP pat matching
(Imperative Statement)	.	S	S	S	S	S	S	S	7	*(Action)*
j := 1;	.	2	.	.	.	.	.	.		pat pointer set to 1
k := 1;	.	3	.	.	.	.	.	.		text pointer set to 1
return("pattern not found");	.	.	.	1	.	.	.	.		return failure message
j := FAIL(j);	.	.	.	.	1	.	.	.		obtain value of shift
return("pattern found at", k-m);	.	.	.	.	.	.	1	.		return success message
k := k+1;	.	.	.	.	.	.	.	1		increment text pointer
j := j+1;	.	.	.	.	.	.	.	2		increment pat pointer
(Conditional Statement)	.	G	G	G	G	G	G	G	.	*(postCondition)*

e) Normalized infoMap: state diagram (unshaded area)

	O	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	7	*(Transition)*
	.	o	.	.	.	.	.	.		compute table and initialize pointers
	.	.	o	.	.	.	.	.		enough text left
	.	.	.	o	.	.	.	.		not enough text left
	.	.	.	.	o	.	.	.		text elem not equal to pat elem
	.	.	.	.	.	o	.	.		text elem equals pat elem
	.	.	.	.	.	.	o	.		all pat chars matched
	.	.	.	.	.	.	.	o		not all pat chars compared
	.	L	L	L	L	L	L	L	5	*(State)*
	.	s	.	.	.	.	.	.		1:.Start
	.	d	s	s	.	.	.	d		2:.Evaluating remaining text
	.	.	d	.	l	s	.	.		3:.Matching text elem with pat elem
	.	.	.	.	.	d	s	s		4:.Testing for pattern occurrence
	.	.	.	d	.	.	d	.		5:.Exit
(Conditional Statement)	.	G	G	G	G	G	G	G	4	*(preCondition)*
k <= n	.	.	t	f	.	.	.	.		enough text left
j>0	.	.	.	.	f	c	.	.		pat pointer > zero
TXT(k) <> PAT(j)	.	.	.	.	f	c	.	.		text elem not equal to pat elem
j = m	.	.	.	.	.	.	t	t		all pat chars compared
(Procedural Statement)	.	S	S	S	S	S	S	S	1	*(Procedure)*
	.	1	.	.	.	.	.	.		compute table for KMP pat matching
(Imperative Statement)	.	S	S	S	S	S	S	S	7	*(Action)*
j := 1;	.	2	.	.	.	.	.	.		pat pointer set to 1
k = 1;	.	3	.	.	.	.	.	.		text pointer set to 1
return("pattern not found");	.	.	.	1	.	.	.	.		return failure message
j := FAIL(j);	.	.	.	.	1	.	.	.		obtain value of shift
return("pattern found at", k-m);	.	.	.	.	.	.	1	.		return success message
k := k+1;	.	.	.	.	.	.	.	1		increment text pointer
j := j+1;	.	.	.	.	.	.	.	2		increment pat pointer
(Conditional Statement)	.	G	G	G	G	G	G	G	.	*(postCondition)*

f) Normalized infoMap to be optimized: shaded area

	O	.	.	.	.	.	.	x	*{Path}*
	S	O	O	O	O	O	O	6	*{Transition}*
	.	o	.	.	.	.	.		compute Fail table and initialize ptrs.
	.	.	o	.	.	.	.		enough text left
	.	.	.	o	.	.	.		not enough left
	.	.	.	.	o	.	.		match at current text location
	.	.	.	.	.	o	.		all pat chars matched
	.	.	.	.	.	.	o		calculate shift for new text location
	.	L	L	L	L	L	L	4	*{State}*
	.	s	.	.	.	.	.		1:.Start
	.	d	s	s	.	.	d		2:.Evaluating remaining text
	.	.	d	.	l	s	s		3:.Matching text elem with pat elem
	.	.	.	d	.	d	.		4:.Exit
{Conditional Statement}	.	G	G	G	G	G	G	3	*{preCondition}*
k <= n	.	.	t	f	.	.	.		enough text left
TXT(k) <> PAT(j)	.	.	.	.	f	f	t		text elem not equal to pat elem
j = m	.	.	.	.	f	t	.		all pat chars compared
{Procedural Statement}	.	S	S	S	S	S	S	1	*{Procedure}*
FAIL()	.	1	.	.	.	.	.		compute table for KMP pat matching
{Imperative Statement}	.	S	S	S	S	S	S	7	*{Action}*
j := 1;	.	2	.	.	.	.	.		pat pointer set to 1
k := 1;	.	3	.	.	.	.	.		text pointer set to 1
return("pattern not found");	.	.	.	1	.	.	.		return failure message
k := k+1;	.	.	.	.	1	.	.		increment text pointer
j := j+1;	.	.	.	.	2	.	.		increment pat pointer
return("pattern found at", k-m);	.	.	.	.	.	1	.		return success message
j := FAIL(j);	.	.	.	.	.	.	1		obtain value of shift
{Conditional Statement}	.	G	G	G	G	G	G	.	*{postCondition}*

g) optimized and documented infoMap

1. Start

compute Fail table and initialize ptrs., CONTINUE at 2

2. Evaluating remaining text

enough text left, CONTINUE at 3

not enough text left, CONTINUE at 4

3. Matching text elem with pat elem

match at current text location, CONTINUE at 3

all pat chars matched, CONTINUE at 4

calculate shift for new text location, CONTINUE at 2

4. Exit

h) Optimized State diagram as Structured English (generated)

1. Start

compute Fail table and initialize ptrs., CONTINUE at 2

FAIL(); j:= 1; k:= 1; goto 2

2. Evaluating remaining text

enough text left, CONTINUE at 3

k<= n -> goto 3

not enough text left, CONTINUE at 4

k > n -> return("pattern not found"); goto 4

3. Matching text elem with pat elem

match at current text location, CONTINUE at 3

TXT(k) = PAT(j) and j<> m -> k:= k+1; j:= j+1; goto 3

all pat chars matched, CONTINUE at 4

j= m and TXT(k) = PAT(j) -> Return ('pattern found at', k-m); goto 4

calculate shift for new text location, CONTINUE at 2

TXT(k) <> PAT(j) -> j:= FAIL(j); goto 2

4. Exit

i) source code (generated)

Fig. 5.1.13 Control flow for Knuth-Morris-Pratt string search algorithm

```
1.  for every character c in the input alphabet do DELTA1(c) := m;
2.  for j := m to 1 by -1 do /* compute DELTA1, initialize DELTA2 */
3.      if(DELTA1(PAT(j)) = m) then DELTA1(PAT(j)) = m-j;
4.      DELTA2(j) := 2*m-j;
    end;
    /* compute DELTA2 */
5.  j := m; t := m+1;
6.  while(j > 0) do
7.      f(j) := t;
8.      while(t <= m and PAT(j) <> PAT(t)) do
9.          DELTA2(t) := min(DELTA2(t), m-j);
10.         t := f(t);
        end;
11.     j := j-1;
12.     t := t-1;
    end;
13. for k := 1 to t do DELTA2(k) := min(DELTA2(k), m+t-k);
    /* The following steps were added by Mehlhorn [4] to ensure */
    /* correct values for DELTA2 in all cases */
14. tp := f(t);
15. while(t <= m) do
16.     while(t <= tp) do
17.         DELTA2(t) := min(DELTA2(t), tp-t+m);
18.         t := t+1;
        end;
19.     tp := f(tp);
    end;
```

a) source code: original

Note: Nos 1 .. 19 in a) are line numbers and should not be confused with states 1 .. 9 of b)

```
1 :   for every character c in the input alphabet do DELTA1(c) := m;
2 :   for j := m to 1 by -1 do /* compute DELTA1, initialize DELTA2 */
3 :         if(DELTA1(PAT(j)) = m) then DELTA1(PAT(j)) = m-j;
            DELTA2(j) := 2*m-j;
      end;
      /* compute DELTA2 */
      j := m; t := m+1;
4 :   while(j > 0) do
            f(j) := t;
5 :         while(t <= m and PAT(j) <> PAT(t)) do
                  DELTA2(t) := min(DELTA2(t), m-j);
                  t := f(t);
            end;
            j := j-1;
            t := t-1;
      end;
6 :   for k := 1 to t do DELTA2(k) := min(DELTA2(k), m+t-k);
      /* The following steps were added by Mehlhorn [4] to ensure */
      /* correct values for DELTA2 in all cases */
      tp := f(t);
7 :   while(t <= m) do
8 :         while(t <= tp) do
                  DELTA2(t) := min(DELTA2(t), tp-t+m);
                  t := t+1;
            end;
            tp := f(tp);
      end;
9 :
```

b) edited source code: line nos removed, states 1 .. 9 identified

	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	*(Path)*
	s	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	16	*(Transition)*
	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		
	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.		
	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.		
	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.		
	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.		
	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		
	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.		
	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.		
	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.		
	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.		
	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.		
	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.		
	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.		
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.		
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.		
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		
	.	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	9	*(State)*
	.	l	s	.	.	.	.	.	.	.	.	.	.	.	.	.	.		1:
	.	.	d	s	s	d	d	.	.	.	.	.	.	.	.	.	.		2:
	.	.	.	.	d	s	s	.	.	.	.	.	.	.	.	.	.		3:
	.	.	.	d	.	.	.	s	s	.	.	.	.	.	.	.	.		4:
	.	.	.	.	.	.	.	.	d	l	s	.	d	.	.	.	.		5:
	.	.	.	.	.	.	.	d	.	.	d	l	s	.	.	.	.		6:
	.	.	.	.	.	.	.	.	.	.	.	.	.	s	s	.	d		7:
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	d	l	s		8:
	.	.	.	.	.	.	.	.	.	.	.	.	.	d	.	.	.		9:
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	8	*(preCondition)*
	.	t	f	.	.	.	.	.	.	.	.	.	.	.	.	.	.		for every character c in the input alphabet
j := m to 1 by -1	.	.	.	f	t	.	.	.	.	.	.	.	.	.	.	.	.		1=< elem <= pattern length
DELTA1(PAT(j)) = m	.	.	.	.	.	t	f	.	.	.	.	.	.	.	.	.	.		elem. of DELTA1 equals pattern length
j > 0	.	.	.	.	.	.	.	f	t	.	.	.	.	.	.	.	.		elem. > zero
t <= m	.	.	.	.	.	.	.	.	.	t	c	.	.	f	t	.	.		elem <= pattern length
PAT(j) <> PAT(t)	.	.	.	.	.	.	.	.	.	t	c	.	.	.	.	.	.		pat elem. j not equal to pat elem t
k := 1 to t	.	.	.	.	.	.	.	.	.	.	.	t	f	.	.	.	.		1 =< current elem. <= t
t <= tp	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	f		right ptr. <= restart ptr.
(Imperative Statement)	.	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	15	*(Action)*
DELTA1(c) := m;	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		set DELTA1 elem. to pat length
DELTA1(PAT(j)) := m j;	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.		compute DELTA1
DELTA2(j) := 2*m-j;	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.		initialize DELTA2
j := m;	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.		set pat ptr to rightmost elem.
t := m+1;	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.		increment mismatch ptr
f(j) := t;	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.		calculate restart ptr.
DELTA2(t) := min(DELTA2(t) m-j);	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.		calculate minimum DELTA2 shift
t := f(t);	.	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.		calculate next t
j := j-1;	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.		decrement left ptr j
t := t-1;	.	.	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.		decrement right ptr t
DELTA2(k) := min(DELTA2(k) m+t-k);	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.		calculate new min DELTA2 shift
tp := f(t);	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.		calculate new restart ptr
DELTA2(t) := min(DELTA2(t) tp-t+m);	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.		verify minimum DELTA2 shift
t := t+1;	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2	.		increment right ptr. t
tp := f(tp),	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1		calculate new restart ptr.
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	.	*(postCondition)*

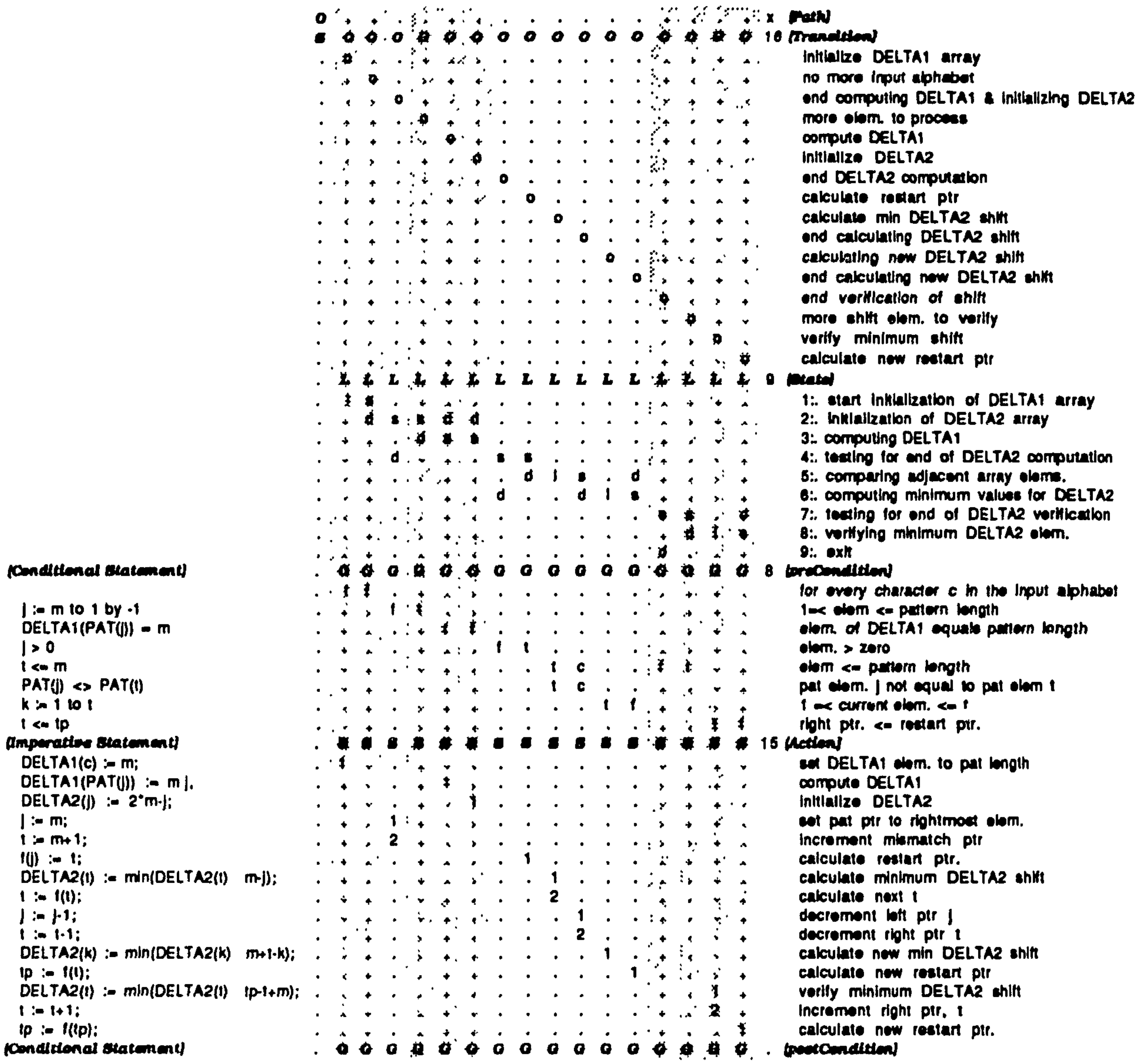
c) Normalised InfoMap: shaded areas to be filled in

	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	(Path)
	s	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	16	(Transition)
	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		initialize DELTA1 array
	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.		no more input alphabet
	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.		end computing DELTA1 & initializing DELTA2
	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.		more elem. to process
	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.		compute DELTA1
	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		initialize DELTA2
	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.		end DELTA2 computation
	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.		calculate restart ptr
	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.		calculate min DELTA2 shift
	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.		end calculating DELTA2 shift
	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.		calculating new DELTA2 shift
	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.		end calculating new DELTA2 shift
	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.		end verification of shift
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.		more shift elem. to verify
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.		verify minimum shift
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		calculate new restart ptr
	.	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	9	(State)
	.	l	s	.	.	.	.	.	.	.	.	.	.	.	.	.	.		1. start initialization of DELTA1 array
	.	.	d	s	s	d	d	.	.	.	.	.	.	.	.	.	.		2. initialization of DELTA2 array
	.	.	.	.	d	s	s	.	.	.	.	.	.	.	.	.	.		3. computing DELTA1
	.	.	.	d	.	.	.	s	s	.	.	.	.	.	.	.	.		4. testing for end of DELTA2 computation
	.	.	.	.	.	.	.	.	d	l	s	.	d	.	.	.	.		5. comparing adjacent array elems.
	.	.	.	.	.	.	.	d	.	.	d	l	s	.	.	.	.		6. computing minimum values for DELTA2
	.	.	.	.	.	.	.	.	.	.	.	.	.	s	s	.	d		7. testing for end of DELTA2 verification
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	d	l	s		8. verifying minimum DELTA2 elem.
	.	.	.	.	.	.	.	.	.	.	.	.	.	d	.	.	.		9. exit
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	8	(preCondition)
	.	t	f	.	.	.	.	.	.	.	.	.	.	.	.	.	.		for every character c in the input alphabet
j := m to 1 by -1	.	.	.	f	t	.	.	.	.	.	.	.	.	.	.	.	.		1=< elem <= pattern length
DELTA1(PAT(j)) = m	.	.	.	.	.	t	f	.	.	.	.	.	.	.	.	.	.		elem. of DELTA1 equals pattern length
j > 0	.	.	.	.	.	.	.	'	t	.	.	.	.	.	.	.	.		elem. > zero
t <= m	.	.	.	.	.	.	.	.	.	t	c	.	.	f	f	.	.		elem <= pattern length
PAT(j) <> PAT(t)	.	.	.	.	.	.	.	.	.	t	c	.	.	.	.	.	.		pat elem. j not equal to pat elem t
k := 1 to t	.	.	.	.	.	.	.	.	.	.	.	t	f	.	.	.	.		1 =< current elem. <= t
t <= tp	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	f		right ptr. <= restart ptr.
(Imperative Statement)	.	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	15	(Action)
DELTA1(c) := m,	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		set DELTA1 elem. to pat length
DELTA1(PAT(j)) := m-j;	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.		compute DELTA1
DELTA2(j) := 2*m-j;	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.		initialize DELTA2
j := m;	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.		set pat ptr to rightmost elem.
t := m+1;	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.		increment mismatch ptr
f(j) := t;	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.		calculate restart ptr.
DELTA2(t) := min(DELTA2(t) m-j);	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.		calculate minimum DELTA2 shift
t := f(t);	.	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.		calculate next t
j := j-1,	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.		decrement left ptr j
t := t-1;	.	.	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.		decrement right ptr t
DELTA2(k) := min(DELTA2(k) m+t k);	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.		calculate new min DELTA2 shift
tp := f(t);	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.		calculate new restart ptr
DELTA2(t) := min(DELTA2(t) tp-t+m),	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.		verify minimum DELTA2 shift
t := t+1;	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2	.		increment right ptr, t
tp := f(tp),	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1		calculate new restart ptr.
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	.	(postCondition)

d) Normalised infoMap: states and transitions added (shaded area)

```
                                               o  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  x  (Path)
                                               #  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  16 (Transition)
                                               .  o  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .     initialize DELTA1 array
                                               .  .  o  .  .  .  .  .  .  .  .  .  .  .  .  .  .     no more input alphabet
                                               .  .  .  o  .  .  .  .  .  .  .  .  .  .  .  .  .     end computing DELTA1 & initializing DELTA2
                                               .  .  .  .  o  .  .  .  .  .  .  .  .  .  .  .  .     more elem. to process
                                               .  .  .  .  .  o  .  .  .  .  .  .  .  .  .  .  .     compute DELTA1
                                               .  .  .  .  .  .  o  .  .  .  .  .  .  .  .  .  .     initialize DELTA2
                                               .  .  .  .  .  .  .  o  .  .  .  .  .  .  .  .  .     end DELTA2 computation
                                               .  .  .  .  .  .  .  .  o  .  .  .  .  .  .  .  .     calculate restart ptr
                                               .  .  .  .  .  .  .  .  .  o  .  .  .  .  .  .  .     calculate min DELTA2 shift
                                               .  .  .  .  .  .  .  .  .  .  o  .  .  .  .  .  .     end calculating DELTA2 shift
                                               .  .  .  .  .  .  .  .  .  .  .  o  .  .  .  .  .     calculating new DELTA2 shift
                                               .  .  .  .  .  .  .  .  .  .  .  .  o  .  .  .  .     end calculating new DELTA2 shift
                                               .  .  .  .  .  .  .  .  .  .  .  .  .  o  .  .  .     end verification of shift
                                               .  .  .  .  .  .  .  .  .  .  .  .  .  .  o  .  .     more shift elem. to verify
                                               .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  o  .     verify minimum shift
                                               .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  o     calculate new restart ptr
                                               .  L  L  L  L  L  L  L  L  L  L  L  L  L  L  L  L  9  (State)
                                               .  l  s  .  .  .  .  .  .  .  .  .  .  .  .  .  .     1:. start initialization of DELTA1 array
                                               .  .  d  s  s  d  d  .  .  .  .  .  .  .  .  .  .     2:. initialization of DELTA2 array
                                               .  .  .  .  d  s  s  .  .  .  .  .  .  .  .  .  .     3:. computing DELTA1
                                               .  .  .  d  .  .  .  s  s  .  .  .  .  .  .  .  .     4:. testing for end of DELTA2 computation
                                               .  .  .  .  .  .  .  .  .  d  l  s  .  d  .  .  .     5:. comparing adjacent array elems.
                                               .  .  .  .  .  .  .  d  .  .  d  l  s  .  .  .  .     6:. computing minimum values for DELTA2
                                               .  .  .  .  .  .  .  .  .  .  .  .  .  s  s  .  d     7:. testing for end of DELTA2 verification
                                               .  .  .  .  .  .  .  .  .  .  .  .  .  .  d  l  s     8:. verifying minimum DELTA2 elem.
                                               .  .  .  .  .  .  .  .  .  .  .  .  .  d  .  .  .     9:. exit
(Conditional Statement)                        .  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  8  (preCondition)
                                               .  t  f  .  .  .  .  .  .  .  .  .  .  .  .  .  .     for every character c in the input alphabet
   j := m to 1 by -1                           .  .  .  f  t  .  .  .  .  .  .  .  .  .  .  .  .     1<= elem <= pattern length
   DELTA1(PAT(j)) = m                          .  .  .  .  .  t  f  .  .  .  .  .  .  .  .  .  .     elem. of DELTA1 equals pattern length
   j > 0                                       .  .  .  .  .  .  .  t  f  .  .  .  .  .  .  .  .     elem. > zero
   t <= m                                      .  .  .  .  .  .  .  .  .  t  c  .  .  f  t  .  .     elem <= pattern length
   PAT(j) <> PAT(t)                            .  .  .  .  .  .  .  .  .  t  c  .  .  .  .  .  .     pat elem. j not equal to pat elem t
   k := 1 to t                                 .  .  .  .  .  .  .  .  .  .  .  t  f  .  .  .  .     1 <= current elem. <= t
   t <= tp                                     .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  t  f     right ptr. <= restart ptr.
(Imperative Statement)                         .  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  15 (Action)
   DELTA1(c) := m;                             .  1  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .     set DELTA1 elem. to pat length
   DELTA1(PAT(j)) := m-j;                      .  .  .  .  .  1  .  .  .  .  .  .  .  .  .  .  .     compute DELTA1
   DELTA2(j) := 2*m-j;                         .  .  .  .  .  .  1  .  .  .  .  .  .  .  .  .  .     initialize DELTA2
   j := m;                                     .  .  .  1  .  .  .  .  .  .  .  .  .  .  .  .  .     set pat ptr to rightmost elem.
   t := m+1;                                   .  .  .  2  .  .  .  .  .  .  .  .  .  .  .  .  .     increment mismatch ptr
   f(j) := t;                                  .  .  .  .  .  .  .  .  1  .  .  .  .  .  .  .  .     calculate restart ptr.
   DELTA2(t) := min(DELTA2(t), m-j);           .  .  .  .  .  .  .  .  .  1  .  .  .  .  .  .  .     calculate minimum DELTA2 shift
   t := f(t);                                  .  .  .  .  .  .  .  .  .  2  .  .  .  .  .  .  .     calculate next t
   j := j-1;                                   .  .  .  .  .  .  .  .  .  .  1  .  .  .  .  .  .     decrement left ptr j
   t := t-1;                                   .  .  .  .  .  .  .  .  .  .  2  .  .  .  .  .  .     decrement right ptr t
   DELTA2(k) := min(DELTA2(k), m+t-k);         .  .  .  .  .  .  .  .  .  .  .  1  .  .  .  .  .     calculate new min DELTA2 shift
   tp := f(t);                                 .  .  .  .  .  .  .  .  .  .  .  .  1  .  .  .  .     calculate new restart ptr
   DELTA2(t) := min(DELTA2(t), tp-t+m);        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  1  .     verify minimum DELTA2 shift
   t := t+1;                                   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  2  .     increment right ptr. t
   tp := f(tp);                                .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  1     calculate new restart ptr.
(Conditional Statement)                        .  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  .  (postCondition)
```

c) Normalized infoMap: state diagram (unshaded area)



f) Normalised InfoMap to be optimized: shaded areas

```
                                            o . . . . . . . . . . . . . x  (Path)
                                            s o o o o o o o o o o o o o 13 (Transition)
                                            . o . . . . . . . . . . . .    initialize DELTA1 array
                                            . . o . . . . . . . . . . .    end computing DELTA1 & initializing DELTA2
                                            . . . o . . . . . . . . . .    compute DELTA1
                                            . . . . o . . . . . . . . .    initialize DELTA2
                                            . . . . . o . . . . . . . .    end DELTA2 computation
                                            . . . . . . o . . . . . . .    calculate restart ptr
                                            . . . . . . . o . . . . . .    calculate min DELTA2 shift
                                            . . . . . . . . o . . . . .    end calculating new DELTA2 shift
                                            . . . . . . . . . o . . . .    calculating new DELTA2 shift
                                            . . . . . . . . . . o . . .    calculate new restart ptr
                                            . . . . . . . . . . . o . .    end verification of shift
                                            . . . . . . . . . . . . o .    verify shift and increment ptr
                                            . . . . . . . . . . . . . o    calculate new restart ptr
                                            . L L L L L L L L L L L L L 7  (State)
                                            . l . . . . . . . . . . . .    1:. start initialization of DELTA1 array
                                            . e s l l . . . . . . . . .    2:. computing DELTA1 & initializing DELTA2
                                            . . d . . s s . . . . . . .    3:. testing for end of DELTA2 computation
                                            . . . . . . d l s . . . . .    4:. comparing adjacent array elems
                                            . . . . . d . . d l s . . .    5:. computing minimum values for DELTA2
                                            . . . . . . . . . . d s f f    6:. verifying DELTA2 minimum values
                                            . . . . . . . . . . . d . .    7:. exit
(Conditional Statement)                     . G G G G G G G G G G G G G 7  (preCondition)
  j := m to 1 by -1                         . . f t t . . . . . . . . .    1=< elem <= pattern length
  DELTA1(PAT(j)) = m                        . . . t f . . . . . . . . .    elem. of DELTA1 equals pattern length
  j > 0                                     . . . . . f t . . . . . . .    elem. > zero
  PAT(j) <> PAT(t)                          . . . . . . . t c . . . . .    elem <= pattern length
  t <= m                                    . . . . . . . t c . . f t t    pat elem. j not equal to pat elem t
  k := 1 to t                               . . . . . . . . . t f . . .    1 =< current elem. <= t
  t <= tp                                   . . . . . . . . . . . . t f    right ptr. <= restart ptr.
(Imperative Statement)                      . S S S S S S S S S S S S S 15 (Action)
  DELTA1(c) := m;                           . 1 . . . . . . . . . . . .    set DELTA1 elem. to pat length
  DELTA1(PAT(j)) := m-j;                    . . . 1 . . . . . . . . . .    compute DELTA1
  DELTA2(j) := 2*m-j;                       . . . . 1 . . . . . . . . .    initialize DELTA2
  j := m;                                   . . 1 . . . . . . . . . . .    set pat ptr to rightmost elem.
  t := m+1;                                 . . 2 . . . . . . . . . . .    increment mismatch ptr
  f(j) := t;                                . . . . . . 1 . . . . . . .    calculate restart ptr.
  DELTA2(t) := min(DELTA2(t)  m-j);         . . . . . . . 1 . . . . . .    calculate minimum DELTA2 shift
  t := f(t);                                . . . . . . . 2 . . . . . .    calculate next t
  j := j-1;                                 . . . . . . . . 1 . . . . .    decrement left ptr,j
  t := t-1;                                 . . . . . . . . 2 . . . . .    decrement right ptr,t
  DELTA2(k) := min(DELTA2(k)  m+t-k);       . . . . . . . . . 1 . . . .    calculate new min DELTA2 shift
  tp := f(t);                               . . . . . . . . . . 1 . . .    calculate new restart ptr
  DELTA2(t) := min(DELTA2(t)  tp-t+m);      . . . . . . . . . . . . 1 .    verify minimum DELTA2 shift
  t := t+1;                                 . . . . . . . . . . . . 2 .    increment right ptr, t
  tp := f(tp);                              . . . . . . . . . . . . . 1    calculate new restart ptr.
(Conditional Statement)                     . G G G G G G G G G G G G G 1  (postCondition)
                                            . 1 . . . . . . . . . . . .    for every character c in the input alphabet
```

g) Optimized and documented infoMap

Note: Steps - optimized state diagram as Structured English (generated) and source code (generated) - are to be done as in Fig. 5.1.12 or Fig. 5.1.13.

Fig. 5.1.14 Control flow for Computing DELTA1 and DELTA2

```
1.    k := m;
2.    while(k <= n) do
3.        j := m; /* j indexes the pattern and k the text */
4.        while(j > 0 and TXT(k) <> PAT(j)) do
5.            j := j-1;
6.            k := k-1;
          end;
7.        if(j = 0) then
8.            return("pattern found at", k+1)
          else /* shift the pattern */
9.            k := k+max(DELTA1(TXT(k)), DELTA2(j));
      end;
10.   return("pattern not found");
```

a) source code: original

```
1:    k := m;
2:    while(k <= n) do
          j := m; /* j indexes the pattern and k the text */
3:        while(j > 0 and TXT(k) <> PAT(j)) do
              j := j-1;
              k := k-1;
          end;
4:        if(j = 0) then
              return("pattern found at", k+1)
          else /* shift the pattern */
              k := k+max(DELTA1(TXT(k)), DELTA2(j));
      end;
      return("pattern not found");
5:
```

b) edited source code: line nos removed, states 1 .. 5 identified

Note: Nos. 1 .. 10 in a) are line numbers and should not be confused with states 1 .. 5 in b)

	O	.	.	.	.	.	.	.	x	***(Path)***
	S	O	O	O	O	O	O	O	7	***(Transition)***
	.	o	.	.	.	.	.	.		
	.	.	o	.	.	.	.	.		
	.	.	.	o	.	.	.	.		
	.	.	.	.	o	.	.	.		
	.	.	.	.	.	o	.	.		
	.	.	.	.	.	.	o	.		
	.	.	.	.	.	.	.	o		
	.	L	L	L	L	L	L	L	5	***(State)***
	.	s	.	.	.	.	.	.		1:
	.	d	s	s	.	.	.	d		2:
	.	.	d	.	l	s	.	.		3:
	.	.	.	.	.	d	s	s		4:
	.	.	.	d	.	.	d	.		5:
(Conditional Statement)	.	G	G	G	G	G	G	G	4	***(preCondition)***
k <= n	.	.	t	f	.	.	.	.		enough text left
j>0	.	.	.	.	t	c	.	.		index to pattern > zero
TXT(k) = PAT(j)	.	.	.	.	t	c	.	.		text elem equals pat elem
j = 0	.	.	.	.	.	.	t	f		all pat chars compared
(Procedural Statement)	.	S	S	S	S	S	S	S	1	***(Procedure)***
	.	1	.	.	.	.	.	.		compute DELTA1 & DELTA2 table
(Imperative Statement)	.	S	S	S	S	S	S	S	7	***(Action)***
k := m;	.	2	.	.	.	.	.	.		set text pointer to m
j := m;	.	.	1	.	.	.	.	.		set pat pointer to m
return("pattern not found");	.	.	.	1	.	.	.	.		return failure message
j := j-1;	.	.	.	.	1	.	.	.		decrement pat pointer
k := k-1;	.	.	.	.	2	.	.	.		decrement text pointer
return("pattern found at", k+1)	.	.	.	.	.	.	1	.		return success message
k := k+max(DELTA1(TXT(k)), DELTA2(j));	.	.	.	.	.	.	.	1		calculate new location
(Conditional Statement)	.	G	G	G	G	G	G	G	.	***(postCondition)***

c) Normalized infoMap: shaded areas to be filled in

	O	.	.	.	.	.	.	.	x	***(Path)***
	S	O	O	O	O	O	O	O	7	***(Transition)***
	.	o	.	.	.	.	.	.		compute shift table and initialize text ptr.
	.	.	o	.	.	.	.	.		location within the text
	.	.	.	o	.	.	.	.		location outside the text
	.	.	.	.	o	.	.	.		continue matching at current text location
	.	.	.	.	.	o	.	.		matching suspended
	.	.	.	.	.	.	o	.		all pat chars matched
	.	.	.	.	.	.	.	o		calculate shift for new text location
	.	L	L	L	L	L	L	L	5	***(State)***
	.	s	.	.	.	.	.	.		1:Start
	.	d	s	s	.	.	.	d		2:Evaluating remaining text
	.	.	d	.	l	s	.	.		3:Matching text elem with pat elem
	.	.	.	.	.	d	s	s		4:Testing for pattern occurrence
	.	.	.	d	.	.	d	.		5:Exit
(Conditional Statement)	.	G	G	G	G	G	G	G	4	***(preCondition)***
k <= n	.	.	t	f	.	.	.	.		enough text left
j>0	.	.	.	.	t	c	.	.		index to pattern > zero
TXT(k) = PAT(j)	.	.	.	.	t	c	.	.		text elem equals pat elem
j = 0	.	.	.	.	.	.	t	f		all pat chars compared
(Procedural Statement)	.	S	S	S	S	S	S	S	1	***(Procedure)***
	.	1	.	.	.	.	.	.		compute DELTA1 & DELTA2 table
(Imperative Statement)	.	S	S	S	S	S	S	S	7	***(Action)***
k := m;	.	2	.	.	.	.	.	.		set text pointer to m
j := m;	.	.	1	.	.	.	.	.		set pat pointer to m
return("pattern not found");	.	.	.	1	.	.	.	.		return failure message
j := j-1;	.	.	.	.	1	.	.	.		decrement pat pointer
k := k-1;	.	.	.	.	2	.	.	.		decrement text pointer
return("pattern found at", k+1)	.	.	.	.	.	.	1	.		return success message
k := k+max(DELTA1(TXT(k)), DELTA2(j));	.	.	.	.	.	.	.	1		calculate new location
(Conditional Statement)	.	G	G	G	G	G	G	G	.	***(postCondition)***

d) Normalized infoMap: states and transitions added (shaded area)

	O	.	.	.	.	.	.	.	x	*{Path}*
	S	O	O	O	O	O	O	O	7	*{Transition}*
	.	o	.	.	.	.	.	.		compute shift table and initialize text ptr.
	.	.	o	.	.	.	.	.		location within the text
	.	.	.	o	.	.	.	.		location outside the text
	.	.	.	.	o	.	.	.		continue matching at current text location
	.	.	.	.	.	o	.	.		matching suspended
	.	.	.	.	.	.	o	.		all pat chars matched
	.	.	.	.	.	.	.	o		calculate shift for new text location
	.	L	L	L	L	L	L	L	5	*{State}*
	.	s	.	.	.	.	.	.		1:.Start
	.	d	s	s	.	.	.	d		2:.Evaluating remaining text
	.	.	d	.	l	s	.	.		3:.Matching text elem with pat elem
	.	.	.	.	.	d	s	s		4:.Testing for pattern occurrence
	.	.	.	d	.	.	d	.		5:.Exit
{Conditional Statement}	.	G	G	G	G	G	G	G	4	*{preCondition}*
k <= n	.	.	t	f	.	.	.	.		enough text left
j>0	.	.	.	.	t	c	.	.		index to pattern > zero
TXT(k) = PAT(j)	.	.	.	.	t	c	.	.		text elem equals pat elem
j = 0	.	.	.	.	.	.	t	f		all pat chars compared
{Procedural Statement}	.	S	S	S	S	S	S	S	1	*{Procedure}*
	.	1	.	.	.	.	.	.		compute DELTA1 & DELTA2 table
{Imperative Statement}	.	S	S	S	S	S	S	S	7	*{Action}*
k := m;	.	2	.	.	.	.	.	.		set text pointer to m
j := m;	.	.	1	.	.	.	.	.		set pat pointer to m
return("pattern not found");	.	.	.	1	.	.	.	.		return failure message
j := j-1;	.	.	.	.	1	.	.	.		decrement pat pointer
k := k-1;	.	.	.	.	2	.	.	.		decrement text pointer
return("pattern found at", k+1)	.	.	.	.	.	.	1	.		return success message
k := k+max(DELTA1(TXT(k)), DELTA2(j));	.	.	.	.	.	.	.	1		calculate new location
{Conditional Statement}	.	G	G	G	G	G	G	G	.	*{postCondition}*

e) Normalized infoMap: state diagram (unshaded area)

	O	.	.	.	.	.	.	.	x	*{Path}*
	S	O	O	O	O	O	O	O	7	*{Transition}*
	.	o	.	.	.	.	.	.		compute shift table and initialize text ptr.
	.	.	o	.	.	.	.	.		location within the text
	.	.	.	o	.	.	.	.		location outside the text
	.	.	.	.	o	.	.	.		continue matching at current text location
	.	.	.	.	.	o	.	.		matching suspended
	.	.	.	.	.	.	o	.		all pat chars matched
	.	.	.	.	.	.	.	o		calculate shift for new text location
	.	L	L	L	L	L	L	L	5	*{State}*
	.	s	.	.	.	.	.	.		1:.Start
	.	d	s	s	.	.	.	d		2:.Evaluating remaining text
	.	.	d	.	l	s	.	.		3:.Matching text elem with pat elem
	.	.	.	.	.	d	s	s		4:.Testing for pattern occurrence
	.	.	.	d	.	.	d	.		5:.Exit
{Conditional Statement}	.	G	G	G	G	G	G	G	4	*{preCondition}*
k <= n	.	.	t	f	.	.	.	.		enough text left
j>0	.	.	.	.	t	c	.	.		index to pattern > zero
TXT(k) = PAT(j)	.	.	.	.	t	c	.	.		text elem equals pat elem
j = 0	.	.	.	.	.	.	t	f		all pat chars compared
{Procedural Statement}	.	S	S	S	S	S	S	S	1	*{Procedure}*
	.	1	.	.	.	.	.	.		compute DELTA1 & DELTA2 table
{Imperative Statement}	.	S	S	S	S	S	S	S	7	*{Action}*
k := m;	.	2	.	.	.	.	.	.		set text pointer to m
j := m;	.	.	1	.	.	.	.	.		set pat pointer to m
return("pattern not found");	.	.	.	1	.	.	.	.		return failure message
j := j-1;	.	.	.	.	1	.	.	.		decrement pat pointer
k := k-1;	.	.	.	.	2	.	.	.		decrement text pointer
return("pattern found at", k+1)	.	.	.	.	.	.	1	.		return success message
k := k+max(DELTA1(TXT(k)), DELTA2(j));	.	.	.	.	.	.	.	1		calculate new location
{Conditional Statement}	.	G	G	G	G	G	G	G	.	*{postCondition}*

f) Normalized infoMap to be optimized: shaded area

	O	.	.	.	.	.	.	x	***{Path}***
	S	*O*	*O*	*O*	*O*	*O*	*O*	6	***{Transition}***
	.	o	.	.	.	.	.		compute shift table and initialize text ptr
	.	.	o	.	.	.	.		location within the text
	.	.	.	o	.	.	.		location outside the text
	.	.	.	.	o	.	.		match at current text location
	.	.	.	.	.	o	.		all pat chars matched
	.	.	.	.	.	.	o		calculate shift for new text location
	.	*L*	*L*	*L*	*L*	*L*	*L*	4	***{State}***
	.	s	.	.	.	.	.		1:.Start
	.	d	s	s	.	.	d		2:.Evaluating remaining text
	.	.	d	.	l	s	s		3:.Matching text elem with pat elem
	.	.	.	d	.	d	.		4:.Exit
{Conditional Statement}	.	*G*	*G*	*G*	*G*	*G*	*G*	3	***{preCondition}***
k <= n	.	.	t	f	.	.	.		enough text left
TXT(k) = PAT(j)	.	.	.	.	t	t	f		text elem equals pat elem
j = 0	.	.	.	.	f	t	.		all pat chars matched
{Procedural Statement}	.	*S*	*S*	*S*	*S*	*S*	*S*	1	***{Procedure}***
	.	1	.	.	.	.	.		compute DELTA1 & DELTA2 table
{Imperative Statement}	.	*S*	*S*	*S*	*S*	*S*	*S*	7	***{Action}***
k := m;	.	2	.	.	.	.	.		set text pointer to m
j := m;	.	.	1	.	.	.	.		set pat pointer to m
return("pattern not found");	.	.	.	1	.	.	.		return failure message
j := j-1;	.	.	.	.	1	.	.		decrement pat pointer
k := k-1;	.	.	.	.	2	.	.		decrement text pointer
return("pattern found at", k+1)	.	.	.	.	.	1	.		return success message
k := k+max(DELTA1(TXT(k)) DELTA2(j));	.	.	.	.	.	.	1		calculate new location
{Conditional Statement}	.	*G*	*G*	*G*	*G*	*G*	*G*	.	***{postCondition}***

g) Optimized and documented infoMap

1. *Start*
   compute shift table and initialize text ptr., CONTINUE at 2

2. *Evaluating remaining text*
   location within the text, CONTINUE at 3
   location outside the text, CONTINUE at 4

3. *Matching text elem with pat elem*
   match at current text location, CONTINUE at 3
   all pat chars matched, CONTINUE at 4
   calculate shift for new text location, CONTINUE at 2

4. *Exit*

h) Optimized State diagram as Structured English (generated)

1. *Start*
   compute shift table and initialize text ptr., CONTINUE at 2
   DELTA1&DELTA2(); k:= m; goto 2

2. *Evaluating remaining text*
   location within the text, CONTINUE at 3
   k<= n -> j:= m; goto 3

   location outside the text, CONTINUE at 4
   k > n -> return("pattern not found"); goto 4

3. *Matching text elem with pat elem*
   match at current text location, CONTINUE at 3
   TXT(k) = PAT(j) and j<> 0 -> j:= j-1; k:= k-1; goto 3

   all pat chars matched, CONTINUE at 4
   TXT(k) = PAT(j) and j= 0 -> Return ('pattern found at', k+1); goto 4

   calculate shift for new text location, CONTINUE at 2
   TXT(k) <> PAT(j) -> k:= k+max(DELTA1(TXT(k)), DELTA2(j)); goto 2

4. *Exit*

i) source code (generated)

Fig. 5.1.15 Control flow for Boyer-Moore string search algorithm

```
/* build_TD1(): constructs the delta 1 shift table from a pattern string */
int    TD1[ASIZE];                       /* output: table for delta 1 shift index */

build_TD1( pstr)                         /* input: the pattern string*/
char *pstr;
{
      int    i;
      char *p;

      for ( i=0; i<ASIZE; i++)           /* initialize the TD1[] table */
            TD1[i] = Plen + 1;
      for (p=pstr; *p; p++)              /* fill in values from pattern string */
            TD1[*p] = Plen - (p - pstr);
}
```

a) source code: original

```
       /* build_TD1(): constructs the delta 1 shift table from a pattern string */
       int    TD1[ASIZE];                  /* output: table for delta 1 shift index */

       build_TD1( pstr)                    /* input: the pattern string*/
       char *pstr;
       {
             int    i;
             char *p;

1:           for ( i=0; i<ASIZE; i++)      /* initialize the TD1[] table */
                   TD1[i] = Plen + 1;
2:           for (p=pstr; *p; p++)         /* fill in values from pattern string */
                   TD1[*p] = Plen - (p - pstr);
       }
3:
```

b) edited source code: states 1 .. 3 identified

	O	.	.	.	.	x	***{Path}***
	S	O	O	O	O	4	***{Transition}***
	.	o	.	.	.		
	.	.	o	.	.		
	.	.	.	o	.		
	.	.	.	.	o		
	.	L	L	L	L	3	***{State}***
	.	l	s	.	.		1:
	.	.	d	l	s		2:
	.	.	.	.	d		3:
{Conditional Statment}	.	G	G	G	G	2	***{preCondition}***
i:= 0 to ASIZE	.	t	f	.	.		elem. within range
p:=pstr; *p; p++	.	.	.	t	f		pointer within range
{Imperative Statment}	.	S	S	S	S	2	***{Action}***
TD1[i] := Plen+1;	.	1	.	.	.		initialize TD1[] table elem
TD1[*p] := Plen - (p - pstr);	.	.	.	1	.		fill in values from pattern string
{Conditional Statment}	.	G	G	G	G	.	***{postCondition}***

c) Normalized infoMap: shaded area to be fill in

	O	.	.	.	.	x	***{Path}***
	S	O	O	O	O	4	***{Transition}***
	.	o	.	.	.		initialize table
	.	.	o	.	.		no more elems. to intialize
	.	.	.	o	.		fill in values
	.	.	.	.	o		end fill in of values
	.	L	L	L	L	3	***{State}***
	.	l	s	.	.		1:Start initialization of table
	.	.	d	l	s		2:Filling in of values into table
	.	.	.	.	d		3:Exit
{Conditional Statment}	.	G	G	G	G	2	***{preCondition}***
i:= 0 to ASIZE	.	t	f	.	.		elem. within range
p:=pstr; *p; p++	.	.	.	t	f		pointer within range
{Imperative Statment}	.	S	S	S	S	2	***{Action}***
TD1[i] := Plen+1;	.	1	.	.	.		initialize TD1[] table elem
TD1[*p] := Plen - (p - pstr);	.	.	.	1	.		fill in values from pattern string
{Conditional Statment}	.	G	G	G	G	.	***{postCondition}***

d) Normalized infoMap: states and transitions added (shaded areas)

	O	.	.	.	.	x	*{Path}*
	S	O	O	O	O	4	*{Transition}*
	x	o	.	.	.		initialize table
	.	.	o	.	.		no more elems. to intialize
	.	.	.	o	.		fill in values
	.	.	.	.	o		end fill in of values
	.	*L*	*L*	*L*	*L*	3	*{State}*
	.	l	s	.	.		1:.Start initialization of table
	x	.	d	l	s		2:.Filling in of values into table
	.	.	.	.	d		3:.Exit
{Conditional Statment}	.	*G*	*G*	*G*	*G*	2	*{preCondition}*
i:= 0 to ASIZE	.	t	f	.	.		elem. within range
p:=pstr; *p; p++	.	.	.	t	f		pointer within range
{Imperative Statment}	.	*S*	*S*	*S*	*S*	2	*{Action}*
TD1[i] := Plen+1;	.	1	.	.	.		initialize TD1[] table elem
TD1[*p] := Plen - (p - pstr);	.	.	.	1	.		fill in values from pattern string
{Conditional Statment}	.	*G*	*G*	*G*	*G*	.	*{postCondition}*

e) Normalized infoMap: state diagram (unshaded area)

	O	.	.	.	.	x	*{Path}*
	S	O	O	O	O	4	*{Transition}*
	.	o	.	.	.		initialize table
	.	.	o	.	.		no more elems. to intialize
	.	.	.	o	.		fill in values
	.	.	.	.	o		end fill in of values
	.	*L*	*L*	*L*	*L*	3	*{State}*
	.	l	s	.	.		1:.Start initialization of table
	.	.	d	l	s		2:.Filling in of values into table
	.	.	.	.	d		3:.Exit
{Conditional Statment}	.	*G*	*G*	*G*	*G*	2	*{preCondition}*
i:= 0 to ASIZE	.	t	f	.	.		elem. within range
p:=pstr; *p; p++	.	.	.	t	f		pointer within range
{Imperative Statment}	.	*S*	*S*	*S*	*S*	2	*{Action}*
TD1[i] := Plen+1;	.	1	.	.	.		initialize the TD1[] table
TD1[*p] := Plen - (p - pstr);	.	.	.	1	.		fill in values from pattern string
{Conditional Statment}	.	*G*	*G*	*G*	*G*	.	*{postCondition}*

f) Normalized infoMap to be optimized: shaded area

	O	.	.	x	*{Path}*
	S	*O*	*O*	2	*{Transition}*
	.	o	.		initialize table
	.	.	o		fill in values
	.	*L*	*L*	3	*{State}*
	.	l	.	.	1:.Start initialization of table
	.	e	l	.	2:.Filling in of values into table
	.	.	e	.	3:.Exit
{Conditional Statment}	.	*G*	*G*	.	*{preCondition}*
{Imperative Statment}	.	*S*	*S*	2	*{Action}*
TD1[i] := Plen+1;	.	1	.	.	initialize TD1[] table elem
TD1[*p] := Plen - (p - pstr);	.	.	1	.	fill in values from pattern string
{Conditional Statment}	.	*G*	*G*	2	*{postCondition}*
i:= 0 to ASIZE	.	t	.	.	elem. within range
p:=pstr; *p; p++	.	.	t	.	pointer within range

g) Optimized and documented InfoMap

Note: Steps - Optimized state diagram as Structured English (generated) and source code (generated) are to be done as in Fig. 5.1.12 or Fig. 5.1.13

Fig. 5.1.16 Control flow for computing table TD1[]

```
{first initialize TD2[] for the minimum matching shift}
TD2[0] := 1; {no match}
lshift := 1;
for j:=1 to (m-1)
do begin {scan further leftward for first matching shift }
      lshift := matchshift(j,lshift);
      TD2[j] := lshift;
end;
{next get correct shift with current char mismatch}
for j:= 0 to (m-1)
do begin
      gotshift := false;
      lshift := TD2[j]; {get initial matching shift}
      while(gotshift = false) and (lshift < m)
      do begin {already have a matching shift}
            {also require current char must not match}
            i := (I[j]-lshift);
            if(i<0) or (p[I[j]] <> p[i])
            then gotshift := true
            else begin {get next matching shift}
                  lshift := lshift + 1;
                  lshift := matchshift(j,lshift)
            end;
      end;
      TD2[j] := lshift; {set final shift}
end;
```

a) source code: original

```
1 :   {first initialize TD2[] for the minimum matching shift}
      TD2[0] := 1; {no match}
      lshift := 1;
2 :   for j:=1 to (m-1)
      do begin {scan further leftward for first matching shift }
            lshift := matchshift(j,lshift);
            TD2[j] := lshift;
      end;
      {next get correct shift with current char mismatch}
3 :   for j:= 0 to (m-1)
      do begin
            gotshift := false;
            lshift := TD2[j]; {get initial matching shift}
4 :         while(gotshift = false) and (lshift < m)
            do begin {already have a matching shift}
                  {also require current char must not match}
                  i := (I[j]-lshift);
5 :               if(i<0) or (p[I[j]] <> p[i])
                  then gotshift := true
                  else begin {get next matching shift}
                        lshift := lshift + 1;
                        lshift := matchshift(j,lshift)
                  end;
            end;
            TD2[j] := lshift; {set final shift}
      end;
6 :
```

b) edited source code: states 1 . . 6 identified

	o												x	*(Path)*
	s	o	o	o	o	o	o	o	o	o	o	o	11	*(Transition)*
		o												
			o											
				o										
					o									
						o								
							o							
								o						
									o					
										o				
											o			
												o		
		L	L	L	L	L	L	L	L	L	L	L	6	*(State)*
		s												1:.
		d	l	s										2:.
				d	s	s		d						3:.
					d		s	s	d	d	d	d		4:.
							d		s	s	s	s		5:.
						d								6:.
(Conditional Statement)		G	G	G	G	G	G	G	G	G	G	G	6	*(preCondition)*
j = 1 to (m-1)			t	t										scan elem. >= 1 and less than m
j = 0 to (m-1)					t	t								scan elem. >= 0 and less than m
gotshift = false							t	c						gotshift equals false
lshift < m							t	c						leftward shift less than m
i<0									t	t				pointer to scan ordering index less than zero
p[l[j]] <> p[i]											t	t		scan ordering elem. not equal pat elem.
(Imperative Statement)		S	S	S	S	S	S	S	S	S	S	S	9	*(Action)*
TD2[0] := 1;		1												let leftward shift be 1
lshift := 1;		2												calculate next leftward shift
lshift := matchshift(j, lshift);			1							2		2		set first elem. of TD2 to 1
TD2[j] := lshift;			2					1						set leftward shift to elem. j in table
gotshift := false;					1									set gotshift to false
lshift := TD2[j];					2									get initial matching shift
i := (l[j]-lshift);							1							calculate ptr. to ordering index
gotshift := true									1		1			set gotshift to true
lshift := lshift + 1;										1		1		increment leftward shift by 1
(Conditional Statement)			G	G	G	G	G	G	G	G	G	G		*(postCondition)*

c) Normalized infoMap: shaded areas to be filled in

	o												x	*(Path)*
	s	o	o	o	o	o	o	o	o	o	o	o	11	*(Transition)*
		o												initialize TD2[] for min matching shift
			o											continue scanning leftward for first matching shift
				o										get correct shift with current char mismatch
					o									continue matching at current pattern location
						o								location outside the pattern (failure)
							o							update subscript i
								o						set final shift and continue at new text location
									o					set gotshift to true
										o				increment leftward pointer
											o			set gotshift to true
												o		increment leftward pointer
		L	L	L	L	L	L	L	L	L	L	L	6	*(State)*
		s												1:.Start
		d	l	s										2:.Scanning
				d	s	s		d						3:.Shifting
					d		s	s	d	d	d	d		4:.Pattern testing
							d		s	s	s	s		5:.next matching shift
						d								6:.Exit
(Conditional Statement)		G	G	G	G	G	G	G	G	G	G	G	6	*(preCondition)*
j:=1 to (m-1)			t	t										scan elem. >= 1 and less than m
j:= 0 to (m 1)					t	t								scan elem. >= 0 and less than m
gotshift = false							t	c						gotshift equals false
lshift < m							t	c						leftward shift less than m
i<0									t	t				pointer to scan ordering index less than zero
p[l[j]] <> p[i]											t	t		scan ordering elem. not equal pat elem.
(Imperative Statement)		S	S	S	S	S	S	S	S	S	S	S	9	*(Action)*
TD2[0] := 1;		1												let leftward shift be 1
lshift := 1;		2												calculate next leftward shift
lshift := matchshift(j, lshift);			1							2		2		set first elem. of TD2 to 1
TD2[j] := lshift;			2					1						set leftward shift to elem. j in table
gotshift := false,					1									set gotshift to false
lshift := TD2[j];					2									get initial matching shift
i := (l[j] lshift);							1							calculate ptr. to ordering index
gotshift := true									1		1			set gotshift to true
lshift := lshift + 1;										1		1		increment leftward shift by 1
(Conditional Statement)			G	G	G	G	G	G	G	G	G	G		*(postCondition)*

d) Normalized infoMap. states and transitions added (shaded areas)

	o	.							.		.	.	x	*(Path)*
	S	*o*	*o*	*o*	*o*	*o*	*o*	*o*	*o*	*o*	*o*	*o*	11	*(Transition)*
		o												initialize TD2[] for min matching shift
	.		o											continue scanning leftward for first matching shift
				o										get correct shift with current char mismatch
					o									continue matching at current pattern location
						o								location outside the pattern (failure)
	.						o							update subscript i
								o						set final shift and continue at new text location
	.								o					set gotshift to true
										o				increment leftward pointer
	.										o			set gotshift to true
												o		increment leftward pointer
		L	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	6	*(State)*
		s												1:.Start
		d	i	s										2:.Scanning
				d	s	s		d						3:.Shifting
					d		s	s	d	d	d	d		4:.Pattern testing
							d		s	s	s	s		5:.next matching shift
						d								6:.Exit
(Conditional Statement)		*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	6	*(preCondition)*
j:=1 to (m-1)			t	t			.							scan elem. >= 1 and less than m
j:= 0 to (m-1)	.				t	t					.	.		scan elem. >= 0 and less than m
gotshift = false							t	c						gotshift equals false
lshift < *m*	.						t	c	.	.	.	.		leftward shift less than m
i<0					.	.	.	.	t	t				pointer to scan ordering index less than zero
p[I[j]] <> p[i]											t	t		scan ordering elem. not equal pat elem.
(Imperative Statement)		*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	9	*(Action)*
TD2[0] := 1;		1	.	.	.	.								let leftward shift be 1
lshift := 1;		2							.		.			calculate next leftward shift
lshift := matchshift(j, lshift);			1							2		2		set first elem. of TD2 to 1
TD2[j] := lshift;		.	2							.	.	.		set leftward shift to elem. j in table
gotshift := false;				.	1									set gotshift to false
lshift := TD2[j];					2									get initial matching shift
i := (I[j]-lshift);						.	1							calculate ptr. to ordering index
gotshift := true									1		1			set gotshift to true
lshift := lshift + 1;								.		1		1		increment leftward shift by 1
(Conditional Statement)			*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*		*(postCondition)*

e) Normalized infoMap. state diagram (unshaded area)

	o								.				x	*(Path)*
	S	*o*	*o*	*o*	*o*	*o*	*o*	*o*	*o*	*o*	*o*	*o*	11	*(Transition)*
		o												initialize TD2[] for min matching shift
			o									.		continue scanning leftward for first matching shift
				o										get correct shift with current char mismatch
					o									continue matching at current pattern location
						o								location outside the pattern (failure)
							o							update subscript i
								o						set final shift and continue at new text location
									o					set gotshift to true
										o				increment leftward pointer
										.	o			set gotshift to true
												o		increment leftward pointer
		L	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	6	*(State)*
		s							.	.	.			1:.Start
		d	i	s					.	.				2:.Scanning
				d	s	s		d	.	.	.	.		3:.Shifting
					d		s	s	d	d	d	d		4:.Pattern testing
							d		s	s	s	s		5:.next matching shift
						d				.				6:.Exit
(Conditional Statement)		*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	6	*(preCondition)*
j:=1 to (m-1)			t	t										scan elem. >= 1 and less than m
j:= 0 to (m-1)					t	t					.			scan elem. >= 0 and less than m
gotshift = false							t	c						gotshift equals false
lshift < m							t	c						leftward shift less than *m*
i<0									t	t				pointer to scan ordering index less than zero
p[I[j]] <> p[i]											t	t		scan ordering elem. not equal pat elem.
(Imperative Statement)		*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	9	*(Action)*
TD2[0] := 1;		.												let leftward shift be 1
lshift := 1;		2												calculate next leftward shift
lshift := matchshift(j, lshift);			1							2		2		set first elem. of TD2 to 1
TD2[j] := lshift;			2					1		.				set leftward shift to elem. j in table
gotshift := false;					1									set gotshift to false
lshift := TD2[j];					2				.					get initial matching shift
i := (I[j]-lshift);							1							calculate ptr. to ordering index
gotshift := true									1		1			set gotshift to true
lshift := lshift + 1;										1		1		increment leftward shift by 1
(Conditional Statement)			*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*		*(postCondition)*

f) Normalized infoMap to be optimized: shaded area

	o	.	.	.	.	.	.	.	.	.	.	x	*(Path)*
	s	o	o	o	o	o	o	o	o	o	o	10	*(Transition)*
	.	o	.	.	.	.	.	.	.	.	.	.	initialize TD2[] for min matching shift
	.	.	o	.	.	.	.	.	.	.	.	.	continue scanning leftward for first matching shift
	.	.	.	o	.	.	.	.	.	.	.	.	get correct shift with current char mismatch
	.	.	.	.	o	.	.	.	.	.	.	.	continue matching at current pattern location
	.	.	.	.	.	o	.	.	.	.	.	.	location outside the pattern (failure)
	.	.	.	.	.	.	o	.	.	.	.	.	calculate pointer to scan ordering array
	.	.	.	.	.	.	.	o	.	.	.	.	set final shift and continue at new text location
	.	.	.	.	.	.	.	.	o	o	.	.	set gotshift to true
	.	.	.	.	.	.	.	.	.	.	o	.	increment leftward shift
	.	L	L	L	L	L	L	L	L	L	L	6	*(State)*
	.	s	.	.	.	.	.	.	.	.	.	.	1:.Start
	.	d	l	s	.	.	.	.	.	.	.	.	2:.Scanning
	.	.	.	d	s	s	.	d	.	.	.	.	3:.Shifting
	.	.	.	.	d	.	s	s	d	d	d	.	4:.Testing for Pattern occurrence
	.	.	.	.	.	.	d	.	s	s	s	.	5:.Calculating next matching shift
	.	.	.	.	.	d	.	.	.	.	.	.	6:.Exit
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	G	6	*(preCondition)*
j:=1 to (m-1)	.	.	t	f	.	.	.	.	.	.	.	.	scan elem. >= 1 and less than m
j:= 0 to (m-1)	.	.	.	.	t	f	.	.	.	.	.	.	scan elem. >= 0 and less than m
gotshift = false	.	.	.	.	.	.	t	c	.	.	.	.	gotshift equals false
lshift < m	.	.	.	.	.	.	t	c	.	.	.	.	leftward shift less than m
i< 0	.	.	.	.	.	.	.	.	t	.	f	.	pointer to scan ordering index less than zero
p I[j]] <> p[i]	.	.	.	.	.	.	.	.	.	t	f	.	scan ordering elem. not equal pat elem.
(Imperative Statement)	.	s	s	s	s	s	s	s	s	s	s	9	*(Action)*
TD2[0] := 1;	.	1	.	.	.	.	.	.	.	.	.	.	set first elem. of TD2 to 1
lshift := 1;	.	2	.	.	.	.	.	.	.	.	.	.	let leftward shift be 1
lshift := matchshift(j, lshift);	.	.	1	.	.	.	.	.	.	2	.	.	calculate next leftward shift
TD2[j] := lshift;	.	.	2	.	.	.	.	1	.	.	.	.	set leftward shift to elem. j in table
gotshift := false;	.	.	.	.	1	.	.	.	.	.	.	.	set gotshift to false
lshift := TD2[j];	.	.	.	.	2	.	.	.	.	.	.	.	get initial matching shift
i := (I[j]-lshift);	.	.	.	.	.	.	1	.	.	.	.	.	calculate ptr. to ordering index
gotshift := true	.	.	.	.	.	.	.	.	1	1	.	.	set gotshift to true
lshift := lshift + 1;	.	.	.	.	.	.	.	.	.	.	1	.	increment leftward shift by 1
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	G	.	*(postCondition)*

g) Optimized and documented infoMap

Note: Steps - Optimized state diagram as Structured English (generated) and source code (generated) - are to be done as in Fig. 5.1.12 or Fig. 5.1.13

Fig. 5.1.17 Control flow for computing table TD2[]

```
{Search for a pattern in text}
gotmatch := false;
k := 0;
while(gotmatch = false) and (k+m<=n)   {enough text is still left}
do begin
      j := 0;                      {j scans the ordered pattern}
      while(j<m) and (p[I[j]] = text[k+I[j]])
            do j:= j+1;
      if(j=m)                      {all pattern chars matched}
      then gotmatch := true
      else begin                   {shift pattern}
            delta1 := TD1[text[k+m]];
            delta2 := TD2[j];
            k := k + max(delta1, delta2);
      end;
end;
if(gotmatch = true)
then Search := k               {pattern match found at text location k}
else Search := (-1)            {no pattern match found in text}
```

a) source code: original

```
1 :   {Search for a pattern in text}
      gotmatch := false;
      k := 0;
2 :   while(gotmatch = false) and (k+m<=n)   {enough text is still left}
      do begin
            j := 0;                      {j scans the ordered pattern}
3 :         while(j<m) and (p[I[j]] = text[k+I[j]])
                  do j:= j+1;
4 :         if(j=m)                      {all pattern chars matched}
            then gotmatch := true
            else begin                   {shift pattern}
                  delta1 := TD1[text[k+m]];
                  delta2 := TD2[j];
                  k := k + max(delta1, delta2);
            end;
      end;
5 :   if(gotmatch = true)
      then Search := k               {pattern match found at text location k}
      else Search := (-1)            {no pattern match found in text}
6 :
```

b) edited source code: states 1 . . 6 identified

	o	.	.	.	.	.	.	.	.	.	x	*(Path)*
	s	*o*	*o*	*o*	*o*	*o*	*o*	*o*	*o*	*o*	9	*(Transition)*
	.	o	.	.	.	.	.	.	.	.		
	.	.	o	.	.	.	.	.	.	.		
	.	.	.	o	.	.	.	.	.	.		
	.	.	.	.	o	.	.	.	.	.		
	.	.	.	.	.	o	.	.	.	.		
	.	.	.	.	.	.	o	.	.	.		
	.	.	.	.	.	.	.	o	.	.		
	.	.	.	.	.	.	.	.	o	.		
	.	.	.	.	.	.	.	.	.	o		
	.	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	6	*(State)*
	.	s	.	.	.	.	.	.	.	.		1.
	.	d	s	s	.	.	d	d	.	.		2.
	.	.	d	.	l	s	.	.	.	.		3.
	.	.	.	.	.	d	s	s	.	.		4.
	.	.	.	d	.	.	.	.	s	s		5.
	.	.	.	.	.	.	.	.	d	d		6.
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	5	***(preCondition)***
gotmatch = false	.	.	t	c	.	.	.	.	f	t		no match
k+m <=n	.	.	t	c	.	.	.	.	.	.		location within text
p[i[j]] = text[k+i[j]]	.	.	.	.	t	c	.	.	.	.		pat elem equals text elem
j<m	.	.	.	.	t	c	.	.	.	.		more pat elem left
j=m	.	.	.	.	.	.	t	f	.	.		all pat chars compared
(Procedural Statement)	.	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	2	***(procedure)***
	.	1	.	.	.	.	.	.	.	.		compute table TD1[]
	.	2	.	.	.	.	.	.	.	.		compute table TD2[]
(Imperative Statement)	.	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	10	***(Action)***
gotmatch := false	.	3	.	.	.	.	.	.	.	.		set gotmatch to false
k:=0;	.	4	.	.	.	.	.	.	.	.		set text pointer to zero
j := 0;	.	.	1	.	.	.	.	.	.	.		set scan pointer to zero
QSearch:=(-1)	.	.	.	1	.	.	.	.	.	1		*no substring match found in text*
j := j+1	.	.	.	.	1	.	.	.	.	.		increment scan pointer by 1
QSearch:=k	.	.	.	.	.	.	.	.	1	.		substring match found at text[k]
gotmatch := *true*	.	.	.	.	.	.	1	.	.	.		set gotmatch to true
delta1 := TD1[text[k+m]];	.	.	.	.	.	.	.	1	.	.		calculate amount of shift p to the right
delta2 := TD2[j];	.	.	.	.	.	.	.	2	.	.		calculate pos in pat where mismatch 1st occur
k := k + max(delta1, delta2);	.	.	.	.	.	.	.	3	.	.		calculate new location
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	.	***(postCondition)***

c) Normalized infoMap: shaded areas to be filled in

	O	.	.	.	.	.	.	.	.	.	x	*(Path)*
	S	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	9	*(Transition)*
	.	o	.	.	.	.	.	.	.	.		compute shift table and initialize text ptr.
	.	.	o	.	.	.	.	.	.	.		location within the text
	.	.	.	o	.	.	.	.	.	.		location outside the text
	.	.	.	.	o	.	.	.	.	.		match at current text location
	.	.	.	.	.	o	.	.	.	.		matching suspended
	.	.	.	.	.	.	o	.	.	.		all pat chars compared
	.	.	.	.	.	.	.	o	.	.		calculate new text location
	.	.	.	.	.	.	.	.	o	.		pattern match occur
	.	.	.	.	.	.	.	.	.	o		no pattern match occur
	.	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	6	*(State)*
	.	s	.	.	.	.	.	.	.	.		1:.Start
	.	d	s	s	.	.	d	d	.	.		2:.Evaluating remaining text
	.	.	d	.	l	s	.	.	.	.		3:.Matching pat elem with text elem
	.	.	.	.	.	d	s	s	.	.		4:.End of pattern testing
	.	.	.	d	.	.	.	.	s	s		5:.Testing for pattern occurrence
	.	.	.	.	.	.	.	.	d	d		6:.Exit
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	5	*(preCondition)*
gotmatch = false	.	.	t	c	.	.	.	.	f	t		no match
k+m <=n	.	.	t	c	.	.	.	.	.	.		location within text
p[l[j]] = text[k+l[j]]	.	.	.	.	t	c	.	.	.	.		pat elem equals text elem
j<m	.	.	.	.	t	c	.	.	.	.		more pat elem left
j=m	.	.	.	.	.	.	t	f	.	.		all pat chars compared
(Procedural Statement)	.	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	2	*(procedure)*
	.	1	.	.	.	.	.	.	.	.		compute table TD1[]
	.	2	.	.	.	.	.	.	.	.		compute table TD2[]
(Imperative Statement)	.	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	10	*(Action)*
gotmatch := false	.	3	.	.	.	.	.	.	.	.		set gotmatch to false
k:=0;	.	4	.	.	.	.	.	.	.	.		set text pointer to zero
j := 0;	.	.	1	.	.	.	.	.	.	.		set scan pointer to zero
QSearch:=(-1)	.	.	.	1	.	.	.	.	.	1		no substring match found in text
j := j+1	.	.	.	.	1	.	.	.	.	.		increment scan pointer by 1
QSearch:=k	.	.	.	.	.	.	.	.	1	.		substring match found at text[k]
gotmatch := true	.	.	.	.	.	.	1	.	.	.		set gotmatch to true
delta1 := TD1[text[k+m]];	.	.	.	.	.	.	.	1	.	.		calculate amount of shift p to the right
delta2 := TD2[j];	.	.	.	.	.	.	.	2	.	.		calculate pos in pat where mismatch 1st occur
k := k + max(delta1, delta2);	.	.	.	.	.	.	.	3	.	.		calculate new location
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	.	*(postCondition)*

d) Normalized infoMap: states and transitions added (shaded areas)

	O										X	{Path}
	#	o	o	o	o	o	o	o	o	o	9	{Transition}
	.	o	.	.	.	.	.	.	.	.		compute shift table and initialize text ptr.
	.	.	o	.	.	.	.	.	.	.		location within the text
	.	.	.	o	.	.	.	.	.	.		location outside the text
	.	.	.	.	o	.	.	.	.	.		match at current text location
	.	.	.	.	.	o	.	.	.	.		matching suspended
	.	.	.	.	.	.	o	.	.	.		all pat chars compared
	.	.	.	.	.	.	.	o	.	.		calculate new text location
	.	.	.	.	.	.	.	.	o	.		pattern match occur
	.	.	.	.	.	.	.	.	.	o		no pattern match occur
	.	L	L	L	L	L	L	L	L	L	6	{State}
	.	s	.	.	.	.	.	.	.	.		1:.Start
	.	d	s	s	.	.	d	d	.	.		2:.Evaluating remaining text
	.	.	d	.	l	s	.	.	.	.		3:.Matching pat elem with text elem
	.	.	.	.	.	d	s	s	.	.		4:.End of pattern testing
	.	.	.	d	.	.	.	.	s	s		5:.Testing for pattern occurrence
	.	.	.	.	.	.	.	.	d	d		6:.Exit
{Conditional Statement}	.	C	C	C	C	C	C	C	C	C	5	{preCondition}
gotmatch = false	.	.	t	c	.	.	.	.	t	t		no match
k+m <=n	.	.	t	c	.	.	.	.	.	.		location within text
p[i[j]] = text[k+i[j]]	.	.	.	.	t	c	.	.	.	.		pat elem equals text elem
j<m	.	.	.	.	t	c	.	.	.	.		more pat elem left
j=m	.	.	.	.	.	.	t	f	.	.		all pat chars compared
{Procedural Statement}	.	#	#	#	#	#	#	#	#	#	2	{procedure}
	.	1	.	.	.	.	.	.	.	.		compute table TD1[]
	.	2	.	.	.	.	.	.	.	.		compute table TD2[]
{Imperative Statement}	.	#	#	#	#	#	#	#	#	#	10	{Action Statement}
gotmatch := false	.	3	.	.	.	.	.	.	.	.		set gotmatch to false
k:=0;	.	4	.	.	.	.	.	.	.	.		set text pointer to zero
j := 0;	.	.	1	.	.	.	.	.	.	.		set scan pointer to zero
QSearch:=(-1)	.	.	.	1	.	.	.	.	.	1		no substring match found in text
j := j+1	.	.	.	.	1	.	.	.	.	.		increment scan pointer by 1
QSearch:=k	.	.	.	.	.	.	.	.	1	.		substring match found at text[k]
gotmatch := true	.	.	.	.	.	.	1	.	.	.		set gotmatch to true
delta1 := TD1[text[k+m]];	.	.	.	.	.	.	.	1	.	.		calculate amount of shift p to the right
delta2 := TD2[j];	.	.	.	.	.	.	.	2	.	.		calculate pos in pat where mismatch 1st occur
k := k + max(delta1, delta2);	.	.	.	.	.	.	.	3	.	.		calculate new location
{Conditional Statement}	.	C	C	C	C	C	C	C	C	C	.	{postCondition}

e) Normalized infoMap: state diagram (unshaded area)

	x	*{Path}*
	9	*{Transition}*
		compute shift table and initialize text ptr.
		location within the text
		location outside the text
		match at current text location
		matching suspended
		all pat chars compared
		calculate new text location
		pattern match occur
		no pattern match occur
	6	*{State}*
		1:.Start
		2:.Evaluating remaining text
		3:.Matching pat elem with text elem
		4:.End of pattern testing
		5:.Testing for pattern occurrence
		6:.Exit
{Conditional Statement}	5	*{preCondition}*
gotmatch = false		no match
k+m <=n		location within text
p[I[j]] = text[k+I[j]]		pat elem equals text elem
j<m		more pat elem left
j=m		all pat chars compared
{Procedural Statement}	2	*{procedure}*
		compute table TD1[]
		compute table TD2[]
{Imperative Statement}	10	*{Action Statement}*
gotmatch := false		set gotmatch to false
k:=0;		set text pointer to zero
j := 0;		set scan pointer to zero
QSearch:=(-1)		no substring match found in text
j := j+1		increment scan pointer by 1
QSearch:=k		substring match found at text[k]
gotmatch := true		set gotmatch to true
delta1 := TD1[text[k+m]];		calculate amount of shift p to the right
delta2 := TD2[j];		calculate pos in pat where mismatch 1st occur
k := k + max(delta1, delta2);		calculate new location
{Conditional Statement}	.	*{postCondition}*

f) Normalized infoMap to be optimized: shaded area

	o	.	.	.	.	.	.	x	*(Path)*
	s	*o*	*o*	*o*	*o*	*o*	*o*	6	*(Transition)*
	.	o	.	.	.	.	.	.	compute shift table and initialize text ptr.
	.	.	o	.	.	.	.	.	location within the text
	.	.	.	o	.	.	.	.	location outside the text
	.	.	.	.	o	.	.	.	match at current text location
	.	.	.	.	.	o	.	.	all pat chars matched
	.	.	.	.	.	.	o	.	calculate new text location
	.	*L*	*L*	*L*	*L*	*L*	*L*	4	*(State)*
	.	s	.	.	.	.	.	.	1:.Start
	.	d	s	s	.	.	d	.	2:.Evaluating remaining text
	.	.	d	.	l	s	s	.	3:.Matching pat elem with text elem
	.	.	.	d	.	d	.	.	4:.Exit
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	3	*(preCondition)*
k+m <=n	.	.	t	f	.	.	.	.	location within text
p[l[j]] = text[k+l[j]]	.	.	.	.	t	t	f	.	pat elem equals text elem
j=m	.	.	.	.	f	t	.	.	all pat chars compared
(Procedural Statement)	.	*S*	*S*	*S*	*S*	*S*	*S*	2	*(procedure)*
TD1[]	.	1	.	.	.	.	.	.	compute table TD1[]
TD2[]	.	2	.	.	.	.	.	.	compute table TD2[]
(Imperative Statement)	.	*S*	*S*	*S*	*S*	*S*	*S*	8	*(Action)*
k:=0;	.	3	.	.	.	.	.	.	set text pointer to zero
j := 0;	.	.	1	.	.	.	.	.	set scan pointer to zero
QSearch:=(-1)	.	.	.	1	.	.	.	.	no substring match found in text
j := j+1	.	.	.	.	1	.	.	.	increment scan pointer by 1
QSearch:=k	.	.	.	.	.	1	.	.	substring match found at text[k]
delta1 := TD1[text[k+m]];	.	.	.	.	.	.	1	.	calculate amount of shift p to the right
delta2 := TD2[j];	.	.	.	.	.	.	2	.	calculate pos where mismatch 1st occur
k := k + max(delta1, delta2);	.	.	.	.	.	.	3	.	calculate new location
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	.	*(postCondition)*

g) Optimized and documented infoMap

1:. Start
compute shift table and initialize text ptr., CONTINUE AT 2

2:. Evaluating remaining text
location within the text, CONTINUE At 3
location outside the text, CONTINUE AT 4

3:. Matching pat elem with text elem
match at current text location, CONTINUE AT 3
all pat chars matched, CONTINUE AT 4
calculate new text location, CONTINUE AT 2

4:. Exit

h) Optimized State diagram as Structured English (generated)

1:. Start

compute shift table and initialize text ptr., CONTINUE AT 2
TD1[]; TD2[]; k := 0;

2:. Evaluating remaining text

location within the text, CONTINUE At 3
k+m <= n -> j:= 0; goto 3

location outside the text, CONTINUE AT 4
k+m > n -> QSearch:= (-1); goto 4

3:. Matching pat elem with text elem

match at current text location, CONTINUE AT 3
p[I[j]] = text[k+I[j]] and j<>m -> j:= j+1; goto 3

all pat chars matched, CONTINUE AT 4
p[I[j]] = text[k+I[j]] and j=m -> QSearch:= k; goto 4

calculate new text location, CONTINUE AT 2
p[I[j]] <> text[k+I[j]] -> delta1:= TD1[text[k+m]];
delta2:= TD2[j]; k:= k+max(delta1, delta2); goto 2

4:. Exit

i) source code (generated)

Fig. 5.1.18 Control flow for a Substring Search algorithm

```
{Quick Search for a string in text}
gotmatch:=false;
k:=0;
while(gotmatch = false) and (k+m <=n)
do begin
      i:=0;                        {i scans the pattern string}
      while(i<m) and (p[i] = text[k+i])
            do i:=i+1;
      if(i=m)                      {all pattern chars matched}
      then gotmatch := true
      else k := k+TD1[text[k+m]]{shift pattern}
end;
if(gotmatch = true)
then QSearch:=k                   {substring match found at text[k]}
else QSearch:=(-1)                {no substring match found in text}
```

a) source code: original

```
1 :   {Quick Search for a string in text}
      gotmatch:=false;
      k:=0;
2 :   while(gotmatch = false) and (k+m <=n)
      do begin
            i:=0;                        {i scans the pattern string}
3 :         while(i<m) and (p[i] = text[k+i])
                  do i:=i+1;
4 :         if(i=m)                      {all pattern chars matched}
            then gotmatch := true
            else k := k+TD1[text[k+m]]{shift pattern}
      end;
5 :   if(gotmatch = true)
      then QSearch:=k                   {substring match found at text[k]}
      else QSearch:=(-1)                {no substring match found in text}
6 :
```

b) edited source code: states 1 . . 6 identified

	O	.	.	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	O	O	9	*(Transition)*
	.	o	.	.	.	.	.	.	.	.		
	.	.	o	.	.	.	.	.	.	.		
	.	.	.	o	.	.	.	.	.	.		
	.	.	.	.	o	.	.	.	.	.		
	.	.	.	.	.	o	.	.	.	.		
	.	.	.	.	.	.	o	.	.	.		
	.	.	.	.	.	.	.	o	.	.		
	.	.	.	.	.	.	.	.	o	.		
	.	.	.	.	.	.	.	.	.	o		
	.	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	6	*(State)*
	.	s	.	.	.	.	.	.	.	.		1
	.	d	s	s	.	.	d	d	.	.		2
	.	.	d	.	l	s	.	.	.	.		3
	.	.	.	.	.	d	s	s	.	.		4
	.	.	.	d	.	.	.	.	s	s		5
	.	.	.	.	.	.	.	.	d	d		6
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	5	*(preCondition)*
gotmatch = false	.	.	t	c	.	.	.	.	f	t		no match
k+m <=n	.	.	t	c	.	.	.	.	.	.		location within text
p[i] = text[k+i]	.	.	.	.	t	c	.	.	.	.		pat elem equals text elem
i<m	.	.	.	.	t	c	.	.	.	.		location outside text
i=m	.	.	.	.	.	.	t	f	.	.		all pat chars compared
(Procedural Statement)	.	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	1	*(Procedure)*
	.	1	.	.	.	.	.	.	.	.		compute table TD1[]
(Imperative Statement)	.	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	8	*(Action)*
gotmatch := false	.	2	.	.	.	.	.	.	.	.		set gotmatch to false
k:=0;	.	3	.	.	.	.	.	.	.	.		set text pointer to zero
i := 0;	.	.	1	.	.	.	.	.	.	.		set scan pointer to zero
gotmatch := true	.	.	.	.	.	.	1	.	.	.		set gotmatch to true
i := i+1	.	.	.	.	1	.	.	.	.	.		increment scan pointer by 1
QSearch:=(-1)	.	.	.	.	.	.	.	.	.	1		no substring match found in text
QSearch:=k	.	.	.	.	.	.	.	.	1	.		substring match found at text[k]
k := k+TD1[text[k+m]]	.	.	.	.	.	.	.	1	.	.		calculate new location
(Conditional Statement)	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	.	*(postCondition)*

c) Normalized infoMap: shaded areas to be filled in

	0	.	.	.	.	.	.	.	.	.	x	*(Path)*
	8	O	O	O	O	O	O	O	O	O	9	*(Transition)*
	.	o	.	.	.	.	.	.	.	.		compute shift table and initialize text ptr.
	.	.	o	.	.	.	.	.	.	.		location within the text
	.	.	.	o	.	.	.	.	.	.		location outside the text
	.	.	.	.	o	.	.	.	.	.		match at current text location
	.	.	.	.	.	o	.	.	.	.		matching suspended
	.	.	.	.	.	.	o	.	.	.		all pat chars compared
	.	.	.	.	.	.	.	o	.	.		calculate new text location
	.	.	.	.	.	.	.	.	o	.		pattern match occur
	.	.	.	.	.	.	.	.	.	o		no pattern match occur
	.	L	L	L	L	L	L	L	L	L	6	*(State)*
	.	s	.	.	.	.	.	.	.	.		1:.Start
	.	d	s	s	.	.	d	d	.	.		2:.Evaluating remaining text
	.	.	d	.	l	s	.	.	.	.		3:.Matching pat elem with text elem
	.	.	.	.	.	d	s	s	.	.		4:.End of pattern testing
	.	.	.	d	.	.	.	.	s	s		5:.Testing for pattern occurrence
	.	.	.	.	.	.	.	.	d	d		6:.Exit
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	5	*(preCondition)*
gotmatch = false	.	.	t	c	.	.	.	.	f	t		no match
k+m <=n	.	.	t	c	.	.	.	.	.	.		location within text
p[i] = text[k+i]	.	.	.	.	t	c	.	.	.	.		pat elem equals text elem
i<m	.	.	.	.	t	c	.	.	.	.		location outside text
i=m	.	.	.	.	.	.	t	f	.	.		all pat chars compared
(Procedural Statement)	.	S	S	S	S	S	S	S	S	S	1	*(Procedure)*
	.	1	.	.	.	.	.	.	.	.		compute table TD1[]
(Imperative Statement)	.	S	S	S	S	S	S	S	S	S	8	*(Action)*
gotmatch := false	.	2	.	.	.	.	.	.	.	.		set gotmatch to false
k:=0;	.	3	.	.	.	.	.	.	.	.		set text pointer to zero
i := 0.	.	.	1	.	.	.	.	.	.	.		set scan pointer to zero
gotmatch := true	.	.	.	.	.	.	1	.	.	.		set *gotmatch* to *true*
i := i+1	.	.	.	.	1	.	.	.	.	.		increment scan pointer by 1
QSearch:=(-1)	.	.	.	.	.	.	.	.	.	1		no substring match found in text
QSearch:=k	.	.	.	.	.	.	.	.	1	.		substring match found at text[k]
k := k+TD1[text[k+m]]	.	.	.	.	.	.	.	1	.	.		calculate new location
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	.	*(postCondition)*

d) Normalized infoMap: states and transitions added (shaded areas)

	0	.	.	.	.	.	.	.	.	.	x	*(Path)*
	8	O	O	O	O	O	O	O	O	O	9	*(Transition)*
	.	o	.	.	.	.	.	.	.	.		compute shift table and initialize text ptr.
	.	.	o	.	.	.	.	.	.	.		location within the text
	.	.	.	o	.	.	.	.	.	.		location outside the text
	.	.	.	.	o	.	.	.	.	.		match at current text location
	.	.	.	.	.	o	.	.	.	.		matching suspended
	.	.	.	.	.	.	o	.	.	.		all pat chars compared
	.	.	.	.	.	.	.	o	.	.		calculate new text location
	.	.	.	.	.	.	.	.	o	.		pattern match occur
	.	.	.	.	.	.	.	.	.	o		no pattern match occur
	.	L	L	L	L	L	L	L	L	L	6	*(State)*
	.	s	.	.	.	.	.	.	.	.		1:.Start
	.	d	s	s	.	.	d	d	.	.		2:.Evaluating remaining text
	.	.	d	.	l	s	.	.	.	.		3:.Matching pat elem with text elem
	.	.	.	.	.	d	s	s	.	.		4:.End of pattern testing
	.	.	.	d	.	.	.	.	s	s		5:.Testing for pattern occurrence
	.	.	.	.	.	.	.	.	d	d		6:.Exit
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	5	*(preCondition)*
gotmatch = false	.	.	t	c	.	.	.	.	f	t		no match
k+m <=n	.	.	t	c	.	.	.	.	.	.		location within text
p[i] = text[k+i]	.	.	.	.	t	c	.	.	.	.		pat elem equals text elem
i<m	.	.	.	.	t	c	.	.	.	.		location outside text
i=m	.	.	.	.	.	.	t	f	.	.		all pat chars compared
(Procedural Statement)	.	S	S	S	S	S	S	S	S	S	1	*(Procedure)*
	.	1	.	.	.	.	.	.	.	.		compute table TD1[]
(Imperative Statement)	.	S	S	S	S	S	S	S	S	S	8	*(Action)*
gotmatch := false	.	2	.	.	.	.	.	.	.	.		set gotmatch to *false*
k:=0;	.	3	.	.	.	.	.	.	.	.		set text pointer to zero
i := 0;	.	.	1	.	.	.	.	.	.	.		set scan pointer to zero
gotmatch := true	.	.	.	.	.	.	1	.	.	.		set gotmatch to true
i := i+1	.	.	.	.	1	.	.	.	.	.		increment scan pointer by 1
QSearch:=(-1)	.	.	.	.	.	.	.	.	.	1		no substring match found in text
QSearch:=k	.	.	.	.	.	.	.	.	1	.		substring match found at text[k]
k := k+TD1[text[k+m]]	.	.	.	.	.	.	.	1	.	.		calculate new location
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	.	*(postCondition)*

e) Normalized infoMap: state diagram (unshaded area)

	O	.	.	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	O	O	9	*(Transition)*
	.	O	.	.	.	.	.	.	.	.		compute shift table and initialize text ptr.
	.	.	O	.	.	.	.	.	.	.		location within the text
	.	.	.	O	.	.	.	.	.	.		location outside the text
	.	.	.	.	O	.	.	.	.	.		match at current text location
	.	.	.	.	.	O	.	.	.	.		matching suspended
	.	.	.	.	.	.	O	.	.	.		all pat chars compared
	.	.	.	.	.	.	.	O	.	.		calculate new text location
	.	.	.	.	.	.	.	.	O	.		pattern match occur
	.	.	.	.	.	.	.	.	.	O		no pattern match occur
	.	L	L	L	L	L	L	L	L	L	6	*(State)*
	.	s	.	.	.	.	.	.	.	.		1:.Start
	.	d	s	s	.	.	d	d	.	.		2:.Evaluating remaining text
	.	.	d	.	l	s	.	.	.	.		3:.Matching pat elem with text elem
	.	.	.	.	.	d	s	s	.	.		4:.End of pattern testing
	.	.	.	d	.	.	.	.	s	s		5:.Testing for pattern occurrence
	.	.	.	.	.	.	.	.	d	d		6:.Exit
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	5	*(preCondition)*
gotmatch = false	.	.	t	c	.	.	.	.	t	t		no match
k+m <=n	.	.	t	c	.	.	.	.	.	.		location within text
p[i] = text[k+i]	.	.	.	.	t	c	.	.	.	.		pat elem equals text elem
i<m	.	.	.	.	t	c	.	.	.	.		location outside text
i=m	.	.	.	.	.	.	t	f	.	.		all pat chars compared
(Procedural Statement)	.	S	S	S	S	S	S	S	S	S	1	*(Procedure)*
	.	1	.	.	.	.	.	.	.	.		compute table TD1[]
(Imperative Statement)	.	S	S	S	S	S	S	S	S	S	8	*(Action)*
gotmatch := false	.	2	.	.	.	.	.	.	.	.		set gotmatch to false
k:=0;	.	3	.	.	.	.	.	.	.	.		set text pointer to zero
i := 0;	.	.	1	.	.	.	.	.	.	.		set scan pointer to zero
gotmatch := true	.	.	.	.	.	.	1	.	.	.		set gotmatch to true
i := i+1	.	.	.	.	1	.	.	.	.	.		increment scan pointer by 1
QSearch:=(-1)	.	.	.	.	.	.	.	.	.	1		no substring match found in text
QSearch:=k	.	.	.	.	.	.	.	.	1	.		substring match found at text[k]
k := k+TD1[text[k+m]]	.	.	.	.	.	.	.	1	.	.		calculate new location
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	.	*(postCondition)*

f) Normalized infoMap to be optimized: shaded area

	O	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	6	*(Transition)*
	.	o	.	.	.	.	.		compute shift table and initialize text ptr.
	.	.	o	.	.	.	.		location within the text
	.	.	.	o	.	.	.		location outside the text
	.	.	.	.	o	.	.		match at current text location
	.	.	.	.	.	o	.		all pat chars matched
	.	.	.	.	.	.	o		calculate new text location
	.	L	L	L	L	L	L	4	*(State)*
		s	.	.	.	.	.		1:.Start
	.	d	s	s	.	.	d		2..Evaluating remaining text
	.	.	d	.	l	s	s		3:.Matching pat elem with text elem
	.	.	.	d	.	d	.		4..Exit
(Conditional Statement)	.	G	G	G	G	G	G	3	*(preCondition)*
k+m <=n	.	.	t	f	.	.	.		location within text
p[i] = text[k+i]	.	.	.	.	t	t	f		pat elem equals text elem
i=m	.	.	.	.	f	t	.		all pat chars compared
(Procedural Statement)	.	S	S	S	S	S	S	1	*(Procedure)*
	.	1	.	.	.	.	.		computing table TD1[]
(Imperative Statement)	.	S	S	S	S	S	S	6	*(Action)*
k:=0;	.	2	.	.	.	.	.		set text pointer to zero
i := 0;	.	.	1	.	.	.	.		set scan pointer to zero
QSearch:=(-1)	.	.	.	1	.	.	.		no substring match found in text
i := i+1	.	.	.	.	1	.	.		increment scan pointer by 1
QSearch:=k	.	.	.	.	.	1	.		substring match found at text[k]
k := k+TD1[text[k+m]]	.	.	.	.	.	.	1		calculate new location
(Conditional Statement)	.	G	G	G	G	G	G	.	*(postCondition)*

g) Optimized and documented infoMap

1:. Start
compute shift table and initialize text ptr., CONTINUE AT 2

2:. Evaluating remaining text
location within the text, CONTINUE AT 3
location outside the text, CONTINUE AT 4

3:. Matching pat elem with text elem
match at current text location, CONTINUE AT 3
all pat chars matched, CONTINUE AT 4
calculate new text location, CONTINUE AT 2

4:. Exit

h) Optimized State diagram as Structured English (generated)

1:. Start
compute shift table and initialize text ptr., CONTINUE AT 2
TD1[]; k:= 0; goto 2

2:. Evaluating remaining text
location within the text, CONTINUE AT 3
k+m <= n -> i:= 0; goto 3

location outside the text, CONTINUE AT 4
k+m > n -> goto 4

3:. Matching pat elem with text elem
match at current text location, CONTINUE AT 3
i< m and p[i] = text[k+1] -> i:= i+1; goto 3

all pat chars matched, CONTINUE AT 4
i= m and p[i] = text[k+1] -> Return ("substring found at",text[k]); goto 4

calculate new text location, CONTINUE AT 2
p[i] = text[k+1] -> k:= k+TD1[text[k+m]]; goto 2

4:. Exit
i) source code (generated)

Fig. 5.1.19 Control flow for a Quick Search algorithm

Appendix IV

Degree-Constrained Minimum Spanning Tree algorithms

This appendix includes the transformation of three algorithms - Primal, Dual and Anneal. The transformed algorithm Dual has been split into two infoMaps because it was too large to fit on a page. States 9..13 of Fig. 5.2.4.3 are grouped together and treated as a procedure which is transformed into an infoMap (see Fig. 5.2.4.4).

Algorithm Primal:

Inputs: $G = (V, E)$, $V = \{1, 2, \ldots, n\}$, $b > 0$.
$\forall e \in E$, $w[e]$ is the weight for edge e.

Definitions: S : set of edges already in DCST. $S \subseteq E$.
$\forall i \in V$, degree$[i]$ is the current degree of i in DCST.

Algorithm:

(1) Use gen_dcst() to generate S for a DCST.
(2) loop n times
  begin
(3)   Let e_{ij} be a random edge in S.
(4)   Let T_i and T_j be the subtrees reachable from i and j if e_{ij} is removed.
(5)   If $\neg\exists e \in E$ s.t. $e \neq e_{ij}$ and e connects T_i and T_j, goto (10).
(6)   loop 20 times
    begin
(7)   Let e_{vw} be a random edge in E and $e_{uw} \neq e_{ij}$.
(8)   If $w[e_{vw}] < w[e_{ij}]$ and degree$[v] < b$ and degree$[w] < b$ then $S = S - \{e_{ij}\} \cup \{e_{vw}\}$, goto (10).
(9)   If degree$[v] < b$ and degree$[w] < b$ then goto (10).
    end {loop (6)}
(10) end {loop (2)}
(11) Return S and its cost.

(Declaration)	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	14	*(data-object/Attribute)*
i	o	.	.	.	.	.	.	.	.	.	.	.	.	.		a node corresponding to j
j	.	o	.	.	.	.	.	.	.	.	.	.	.	.		a node in V
W	.	.	o	.	.	.	.	.	.	.	.	.	.	.		2-dimensional matrix of weights
S	.	.	.	o	.	.	.	.	.	.	.	.	.	.		a subset of edges in a graph
e	.	.	.	.	o	.	.	.	.	.	.	.	.	.		an edge
b	.	.	.	.	.	o	.	.	.	.	.	.	.	.		max. edges connected to a node
cost of S	.	.	.	.	.	.	o	.	.	.	.	.	.	.		summation of weights in a tree
n	.	.	.	.	.	.	.	o	.	.	.	.	.	.		no. of nodes in the graph
E	.	.	.	.	.	.	.	.	o	.	.	.	.	.		a set of edges
T	.	.	.	.	.	.	.	.	.	o	.	.	.	.		a subset of the nodes in a graph
w	.	.	.	.	.	.	.	.	.	.	o	.	.	.		a node of T
v	.	.	.	.	.	.	.	.	.	.	.	o	.	.		a node of T
d[]	.	.	.	.	.	.	.	.	.	.	.	.	o	.		degree of node
DCST	.	.	.	.	.	.	.	.	.	.	.	.	.	o		Degree-constrained Spanning Tree
(Conditional Statement)	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	7	*(preCondition)*
loop <= n	.	.	.	.	.	.	.	i	.	.	.	.	.	.		loop n times
	i	i	.	.	i	.	.	.	.	.	.	.	.	.		eij removed
	i	i	.	.	i	.	.	.	.	.	.	.	.	.		e <> eij and e connects Ti & Tj
loop <= 20	.	.	.	.	.	.	.	.	.	.	.	.	.	.		loop 20 times
W[evw] < W[eij]	i	i	i	.	i	.	.	.	.	.	i	i	.	.		wt. of random edge vw < edge ij
d[v] < b	.	.	.	.	.	i	.	.	.	.	.	i	i	.		degree of node v less than b
d[w] < b	.	.	.	.	.	i	.	.	.	.	i	.	i	.		degree of node w less than b
(Procedural Statement)	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	1	*(Procedure)*
	.	.	.	.	.	.	.	.	.	.	.	.	.	i		generate DCST
(Imperative Statement)	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	5	*(Action)*
	.	.	.	o	.	.	o	.	.	.	.	.	.	.		Return S and its cost
	i	i	.	i	o	.	.	.	.	.	.	.	.	.		let eij be a random edge in S
	i	i	.	.	.	.	.	.	.	o	.	.	.	.		let Ti & Tj be the subtrees reacheable from i and j
	.	.	.	.	o	.	.	.	i	.	i	i	.	.		let evw be a random & not equal to eij
S := S - {eij} U {evw}	i	i	.	b	i	.	.	.	.	.	i	i	.	.		update set of edges in DCST
(Conditional Statement)	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	.	*(postCondition)*

Fig. 5.2.3.4 Data flow for Primal algorithm

a) source code: original (see page 137)

Inputs: $G = (V, E)$, $V = \{1, 2, \ldots, n\}$, $b > 0$.
$\forall e \in E$, $w[e]$ is the weight for edge e.

Definitions: S : set of edges already in DCST. $S \subseteq E$.
$\forall i \in V$, degree$[i]$ is the current degree of i in DCST.

Algorithm:

1: Use gen_dcst() to generate S for a DCST.
2: loop n times
    begin
        Let e_{ij} be a random edge in S.
3:      Let T_i and T_j be the subtrees reachable from
            i and j if e_{ij} is removed.
4:      If $\neg\exists e \in E$ s.t. $e \neq e_{ij}$ and e connects
            T_i and T_j, goto 2.
5:      loop 20 times
          begin
            Let e_{vw} be a random edge in E and $e_{uw} \neq e_{ij}$.
6:          If $w[e_{vw}] < w[e_{ij}]$ and degree$[v] < b$ and degree$[w] < b$
              then $S = S - \{e_{ij}\} \cup \{e_{vw}\}$, goto 2.
7:          If degree$[v] < b$ and degree$[w] < b$ then goto 2.
          end {loop 5}
    end {loop 2}
8: Return S and its cost.

b) edited source code: line nos removed, states 1..8 identified

	O	.	.	.	.	.	.	.	.	.	.	.	.	.	x	{Path}
	S	O	O	O	O	O	O	O	O	O	O	O	O	O	13	{Transition}
	.	o	.	.	.	.	.	.	.	.	.	.	.	.		
	.	.	o	.	.	.	.	.	.	.	.	.	.	.		
	.	.	.	o	.	.	.	.	.	.	.	.	.	.		
	.	.	.	.	o	.	.	.	.	.	.	.	.	.		
	.	.	.	.	.	o	.	.	.	.	.	.	.	.		
	.	.	.	.	.	.	o	.	.	.	.	.	.	.		
	.	.	.	.	.	.	.	o	.	.	.	.	.	.		
	.	.	.	.	.	.	.	.	o	.	.	.	.	.		
	.	.	.	.	.	.	.	.	.	o	.	.	.	.		
	.	.	.	.	.	.	.	.	.	.	o	.	.	.		
	.	.	.	.	.	.	.	.	.	.	.	o	.	.		
	.	.	.	.	.	.	.	.	.	.	.	.	o	.		
	.	.	.	.	.	.	.	.	.	.	.	.	.	o		
	.	L	L	L	L	L	L	L	L	L	L	L	L	L	8	{State}
	.	s	.	.	.	.	.	.	.	.	.	.	.	.		1:
	.	d	s	s	.	.	d	.	d	.	d	.	d	.		2:
	.	.	.	d	s	s	.	.	.	.	.	.	.	.		3:
	.	.	.	.	d	d	s	s	.	.	.	.	.	.		4:
	.	.	.	.	.	.	.	d	s	s	.	.	.	d		5:
	.	.	.	.	.	.	.	.	.	d	s	s	.	.		6:
	.	.	.	.	.	.	.	.	.	.	.	d	s	s		7:
	.	.	d	.	.	.	.	.	.	.	.	.	.	.		8:
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	G	G	G	9	*{preCondition}*
loop <= n	.	.	f	t	.	.	.	.	.	.	.	.	.	.		loop n times
	.	.	.	.	t	f	.	.	.	.	.	.	.	.		eij removed
	.	.	.	.	.	.	t	f	.	.	.	.	.	.		e <> eij and e connects Ti & Tj
loop <= 20	.	.	.	.	.	.	.	.	f	t	.	.	.	.		loop 20 times
W[evw] < W[eij]	.	.	.	.	.	.	.	.	.	.	t	c	.	.		wt. of random edge vw < edge ij
d[v] < b	.	.	.	.	.	.	.	.	.	.	t	c	.	.		degree of node v less than b
d[w] < b	.	.	.	.	.	.	.	.	.	.	t	c	.	.		degree of node w less than b
d[v] < b	.	.	.	.	.	.	.	.	.	.	.	.	t	c		degree of node v less than b
d[w] < b	.	.	.	.	.	.	.	.	.	.	.	.	t	c		degree of node w less than b
{Procedural Statement}	.	S	S	S	S	S	S	S	S	S	S	S	S	S	1	*{Procedure}*
	.	1	.	.	.	.	.	.	.	.	.	.	.	.		generate DCST
{Imperative Statement}	.	S	S	S	S	S	S	S	S	S	S	S	S	S	5	*{Action}*
	.	.	1	.	.	.	.	.	.	.	.	.	.	.		Return S and its cost
	.	.	.	1	.	.	.	.	.	.	.	.	.	.		let eij be a random edge in S
	.	.	.	.	1	.	.	.	.	.	.	.	.	.		let Ti & Tj be the subtrees
																reacheable from i and j
	.	.	.	.	.	.	.	.	.	1	.	.	.	.		let evw be a random & *not equal to* eij
S := S - {eij} U {evw}	.	.	.	.	.	.	.	.	.	.	1	.	.	.		update subset of edges
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	G	G	G	.	*{postCondition}*

c) Normalizea ..goMap: shaded area to be filled in

	O	.	.	.	.	.	.	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	O	O	O	O	O	O	13	*(Transition)*
	.	o	.	.	.	.	.	.	.	.	.	.	.	.		DCST generated
	.	.	o	.	.	.	.	.	.	.	.	.	.	.		no more nodes to examine
	.	.	.	o	.	.	.	.	.	.	.	.	.	.		more nodes to examine
	.	.	.	.	o	.	.	.	.	.	.	.	.	.		edge ij removed
	.	.	.	.	.	o	.	.	.	.	.	.	.	.		edge ij not removed
	.	.	.	.	.	.	o	.	.	.	.	.	.	.		a different edge from eij exists
	.	.	.	.	.	.	.	o	.	.	.	.	.	.		no different edge from eij exist
	.	.	.	.	.	.	.	.	o	.	.	.	.	.		no more random edge to generate
	.	.	.	.	.	.	.	.	.	o	.	.	.	.		more random edge to generate
	.	.	.	.	.	.	.	.	.	.	o	.	.	.		update set of edges in DCST
	.	.	.	.	.	.	.	.	.	.	.	o	.	.		no update to set of edges in DCST
	.	.	.	.	.	.	.	.	.	.	.	.	o	.		degree of nodes v & w less than b
	.	.	.	.	.	.	.	.	.	.	.	.	.	o		degree of v & w not less than b
	.	L	L	L	L	L	L	L	L	L	L	L	L	L	8	*(State)*
	.	s	.	.	.	.	.	.	.	.	.	.	.	.		1:.Start generation of DCST
	.	d	s	s	.	.	d	.	d	.	d	.	d	.		2:.return of S and its cost
	.	.	.	d	s	s	.	.	.	.	.	.	.	.		3:.Identifying subtrees
	.	.	.	.	d	d	s	s	.	.	.	.	.	.		4:.testing of edges
	.	.	.	.	.	.	.	d	s	s	.	.	.	d		5:.selecting a random edge
	.	.	.	.	.	.	.	.	.	d	s	s	.	.		6:.updating set of edges in DCST
	.	.	.	.	.	.	.	.	.	.	.	d	s	s		7:.testing degree of node
	.	.	d	.	.	.	.	.	.	.	.	.	.	.		8:.Exit
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	G	G	G	G	9	*(preCondition)*
loop <= n	.	.	f	t	.	.	.	.	.	.	.	.	.	.		loop n times
	.	.	.	.	t	f	.	.	.	.	.	.	.	.		eij removed
	.	.	.	.	.	.	t	f	.	.	.	.	.	.		e <> eij and e connects Ti & Tj
loop <= 20	.	.	.	.	.	.	.	.	f	t	.	.	.	.		loop 20 times
W[evw] < W[eij]	.	.	.	.	.	.	.	.	.	.	t	c	.	.		wt. of random edge vw < edge ij
d[v] < b	.	.	.	.	.	.	.	.	.	.	t	c	.	.		degree of node v less than b
d[w] < b	.	.	.	.	.	.	.	.	.	.	t	c	.	.		degree of node w less than b
d[v] < b	.	.	.	.	.	.	.	.	.	.	.	.	t	c		degree of node v less than b
d[w] < b	.	.	.	.	.	.	.	.	.	.	.	.	t	c		degree of node w less than b
(Procedural Statement)	.	S	S	S	S	S	S	S	S	S	S	S	S	S	1	*(Procedure)*
	.	1	.	.	.	.	.	.	.	.	.	.	.	.		generate DCST
(Imperative Statement)	.	S	S	S	S	S	S	S	S	S	S	S	S	S	5	*(Action)*
	.	.	1	.	.	.	.	.	.	.	.	.	.	.		Return S and its cost
	.	.	.	1	.	.	.	.	.	.	.	.	.	.		let eij be a random edge in S
	.	.	.	.	1	.	.	.	.	.	.	.	.	.		let Ti & Tj be the subtrees reacheable from i and j
	.	.	.	.	.	.	.	.	.	1	.	.	.	.		let evw be a random & not equal to eij
S := S - {eij} U {evw}	.	.	.	.	.	.	.	.	.	.	1	.	.	.		update set of edges in DCST
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	G	G	G	G	.	*(postCondition)*

d) Normalized InfoMap: states and transitions added (shaded area)

	O	.	x	.	.	.	x	.	.	x	.	.	.	x	x	{Path}
	S	O	O	O	O	O	O	O	O	O	O	O	O	O	13	{Transition}
	.	o	.	.	.	.	.	.	.	.	.	.	.	.		DCST generated
	.	.	o	.	.	.	.	.	.	.	.	.	.	.		no more nodes to examine
	.	.	.	o	.	.	.	.	.	.	.	.	.	.		more nodes to examine
	.	.	.	.	o	.	.	.	.	.	.	.	.	.		edge ij removed
	.	.	.	.	.	o	.	.	.	.	.	.	.	.		edge ij not removed
	.	.	.	.	.	.	o	.	.	.	.	.	.	.		a different edge from eij exists
	.	.	.	.	.	.	.	o	.	.	.	.	.	.		no different edge from eij exist
	.	.	.	.	.	.	.	.	o	.	.	.	.	.		no more random edge to generate
	.	.	.	.	.	.	.	.	.	o	.	.	.	.		more random edge to generate
	.	.	.	.	.	.	.	.	.	.	o	.	.	.		update set of edges in DCST
	.	.	.	.	.	.	.	.	.	.	.	o	.	.		no update to set of edges in DCST
	.	.	.	.	.	.	.	.	.	.	.	.	o	.		degree of nodes v & w less than b
	.	.	.	.	.	.	.	.	.	.	.	.	.	o		degree of v & w not less than b
	.	L	L	L	L	L	L	L	L	L	L	L	L	L	8	{State}
	.	s	.	.	.	.	.	.	.	.	.	.	.	.		1:.Start generation of DCST
	.	d	s	s	.	.	d	.	d	.	d	.	d	.		2:.return of S and its cost
	.	.	.	d	s	s	.	.	.	.	.	.	.	.		3:.identifying subtrees
	.	.	.	.	d	d	s	s	.	.	.	.	.	.		4:.testing of edges
	.	.	.	.	.	.	.	d	s	s	.	.	.	d		5:.selectimg a random edge
	.	.	.	.	.	.	.	.	.	d	s	s	.	.		6:.updating set of edges in DCST
	.	.	.	.	.	.	.	.	.	.	.	d	s	s		7:.testing degree of node
	.	.	d	.	.	.	.	.	.	.	.	.	.	.		8:.Exit
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	G	G	G	9	{preCondition}
loop <= n	.	.	f	t	.	.	.	.	.	.	.	.	.	.		loop n times
	.	.	.	.	t	f	.	.	.	.	.	.	.	.		eij removed
	.	.	.	.	.	.	t	f	.	.	.	.	.	.		e <> eij and e connects Ti & Tj
loop <= 20	.	.	.	.	.	.	.	.	f	t	.	.	.	.		loop 20 times
W[evw] < W[eij]	.	.	.	.	.	.	.	.	.	.	t	c	.	.		wt. of random edge vw < edge ij
d[v] < b	.	.	.	.	.	.	.	.	.	.	t	c	.	.		degree of node v less than b
d[w] < b	.	.	.	.	.	.	.	.	.	.	t	c	.	.		degree of node w less than b
d[v] < b	.	.	.	.	.	.	.	.	.	.	.	.	t	c		degree of node v less than b
d[w] < b	.	.	.	.	.	.	.	.	.	.	.	.	t	c		degree of node w less than b
{Procedural Statement}	.	S	S	S	S	S	S	S	S	S	S	S	S	S	1	{Procedure}
	.	1	.	.	.	.	.	.	.	.	.	.	.	.		generate DCST
{Imperative Statement}	.	S	S	S	S	S	S	S	S	S	S	S	S	S	5	{Action}
	.	.	1	.	.	.	.	.	.	.	.	.	.	.		Return S and its cost
	.	.	.	1	.	.	.	.	.	.	.	.	.	.		let eij be a random edge in S
	.	.	.	.	1	.	.	.	.	.	.	.	.	.		let Ti & Tj be the subtrees reacheable from i and j
	.	.	.	.	.	.	.	.	.	1	.	.	.	.		let evw be a random & not equal to eij
S := S - {eij} U {evw}	.	.	.	.	.	.	.	.	.	.	1	.	.	.		update set of edges in DCST
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	G	G	G	.	{postCondition}

e) Normalized infoMap: state diagram (unshaded area)

	O	.	.	.	.	.	.	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	O	O	O	O	O	D	13	*(Transition)*
	.	o	.	.	.	.	.	.	.	.	.	.	.	.		DCST generated
	.	.	o	.	.	.	.	.	.	.	.	.	.	.		no more nodes to examine
	.	.	.	o	.	.	.	.	.	.	.	.	.	.		more nodes to examine
	.	.	.	.	o	.	.	.	.	.	.	.	.	.		edge ij removed
	.	.	.	.	.	o	.	.	.	.	.	.	.	.		edge ij not removed
	.	.	.	.	.	.	o	.	.	.	.	.	.	.		a different edge from eij exists
	.	.	.	.	.	.	.	o	.	.	.	.	.	.		no different edge from eij exist
	.	.	.	.	.	.	.	.	o	.	.	.	.	.		no more random edge to generate
	.	.	.	.	.	.	.	.	.	o	.	.	.	.		more random edge to generate
	.	.	.	.	.	.	.	.	.	.	o	.	.	.		update set of edges in DCST
	.	.	.	.	.	.	.	.	.	.	.	o	.	.		no update to set of edges in DCST
	.	.	.	.	.	.	.	.	.	.	.	.	o	.		degree of nodes v & w less than b
	.	.	.	.	.	.	.	.	.	.	.	.	.	o		degree of v & w not less than b
	.	L	L	L	L	L	L	L	L	L	L	L	L	L	8	*(State)*
	.	s	.	.	.	.	.	.	.	.	.	.	.	.		1:.Start generation of DCST
	.	d	s	s	.	.	d	.	d	.	d	.	d	.		2:.return of S and its cost
	.	.	.	d	s	s	.	.	.	.	.	.	.	.		3:.identifying subtrees
	.	.	.	.	d	d	s	s	.	.	.	.	.	.		4:.testing of edges
	.	.	.	.	.	.	.	d	s	s	.	.	.	d		5:.selectimg a random edge
	.	.	.	.	.	.	.	.	.	d	s	s	.	.		6:.updating set of edges in DCST
	.	.	.	.	.	.	.	.	.	.	.	d	s	s		7:.testing degree of node
	.	.	d	.	.	.	.	.	.	.	.	.	.	.		8:.Exit
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	G	G	G	G	9	*(preCondition)*
loop <= n	.	.	f	t	.	.	.	.	.	.	.	.	.	.		loop n times
	.	.	.	.	t	f	.	.	.	.	.	.	.	.		eij removed
	.	.	.	.	.	.	t	f	.	.	.	.	.	.		e <> eij and e connects Ti & Tj
loop <= 20	.	.	.	.	.	.	.	.	f	t	.	.	.	.		loop 20 times
W[evw] < W[eij]	.	.	.	.	.	.	.	.	.	.	t	c	.	.		wt. of random edge vw < edge ij
d[v] < b	.	.	.	.	.	.	.	.	.	.	t	c	.	.		degree of node v less than b
d[w] < b	.	.	.	.	.	.	.	.	.	.	t	c	.	.		degree of node w less than b
d[v] < b	.	.	.	.	.	.	.	.	.	.	.	.	t	c		degree of node v less than b
d[w] < b	.	.	.	.	.	.	.	.	.	.	.	.	t	c		degree of node w less than b
(Procedural Statement)	.	S	S	S	S	S	S	S	S	S	S	S	S	S	1	*(Procedure)*
	.	1	.	.	.	.	.	.	.	.	.	.	.	.		generate DCST
(Imperative Statement)	.	S	S	S	S	S	S	S	S	S	S	S	S	S	5	*(Action)*
	.	.	1	.	.	.	.	.	.	.	.	.	.	.		Return S and its cost
	.	.	.	1	.	.	.	.	.	.	.	.	.	.		let eij be a random edge in S
	.	.	.	.	1	.	.	.	.	.	.	.	.	.		let Ti & Tj be the subtrees
																reacheable from i and j
	.	.	.	.	.	.	.	.	.	1	.	.	.	.		let evw be a random & not equal to eij
S := S - {eij} U {evw}	.	.	.	.	.	.	.	.	.	.	1	.	.	.		update set of edges in DCST
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	G	G	G	G	.	*(postCondition)*

f) Normalized infoMap to be optimized: shaded area

	O	.	.	.	.	.	.	.	.	.	.	.	.	x	*{Path}*
	S	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	12	*{Transition}*
	.	o	.	.	.	.	.	.	.	.	.	.	.		DCST generated
	.	.	o	.	.	.	.	.	.	.	.	.	.		no more nodes to examine
	.	.	.	o	.	.	.	.	.	.	.	.	.		more nodes to examine
	.	.	.	.	o	.	.	.	.	.	.	.	.		edge ij removed
	.	.	.	.	.	o	.	.	.	.	.	.	.		edge ij not removed
	.	.	.	.	.	.	o	.	.	.	.	.	.		a different edge from eij exists
	.	.	.	.	.	.	.	c	.	.	.	.	.		no different edge from eij exist
	.	.	.	.	.	.	.	.	o	.	.	.	.		no more random edge to generate
	.	.	.	.	.	.	.	.	.	o	.	.	.		more random edge to generate
	.	.	.	.	.	.	.	.	.	.	o	.	.		update set of edges in DCST
	.	.	.	.	.	.	.	.	.	.	.	o	.		end selection of random edge
	.	.	.	.	.	.	.	.	.	.	.	.	o		continue selection of random edge
	.	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	7	*{State}*
	.	s	.	.	.	.	.	.	.	.	.	.	.		1:.Start generation of DCST
	.	d	s	s	.	.	d	.	d	.	d	d	.		2:.examining all nodes of graph
	.	.	.	d	s	s	.	.	.	.	.	.	.		3:.identifying subtrees
	.	.	.	.	d	d	s	s	.	.	.	.	.		4:.testing of edges
	.	.	.	.	.	.	.	d	s	s	.	.	d		5:.selectmg a random edge
	.	.	.	.	.	.	.	.	.	d	s	s	s		6:.updating set of edges in DCST
	.	.	d	.	.	.	.	.	.	.	.	.	.		7:.Exit
{Conditional Statement}	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	7	*{preCondition}*
loop <= n	.	.	f	t	.	.	.	.	.	.	.	.	.		loop n times
	.	.	.	.	t	f	.	.	.	.	.	.	.		eij removed
	.	.	.	.	.	.	t	f	.	.	.	.	.		e <> eij and e connects Ti & Tj
loop <= 20	.	.	.	.	.	.	.	.	f	t	.	.	.		loop 20 times
W[evw] < W[eij]	.	.	.	.	.	.	.	.	.	.	t	.	.		wt. of random edge vw < edge ij
d[v] < b	.	.	.	.	.	.	.	.	.	.	t	t	c		degree of node v less than b
d[w] < b	.	.	.	.	.	.	.	.	.	.	t	t	c		degree of node w less than b
{Procedural Statement}	.	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	1	*{Procedure}*
	.	1	.	.	.	.	.	.	.	.	.	.	.		generate DCST
{Imperative Statement}	.	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	5	*{Action}*
	.	.	1	.	.	.	.	.	.	.	.	.	.		Return S and its cost
	.	.	.	1	.	.	.	.	.	.	.	.	.		let eij be a random edge in S
	.	.	.	.	1	.	.	.	.	.	.	.	.		let Ti & Tj be the subtrees
															reacheable from i and j
	.	.	.	.	.	.	.	.	.	1	.	.	.		let evw be random & not equal to eij
S := S - {eij} U {evw}	.	.	.	.	.	.	.	.	.	.	1	.	.		update set of edges in DCST
{Conditional Statement}	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	.	*{postCondition}*

g) Optimized and documented InfoMap

Note: Steps - Optimized state diagram as Structured English (generated) and source code (generated) - are to be done as in Fig. 5.2.4.1

Fig. 5.2.4.2 Control flow for Primal algorithm

Dual algorithm:

Inputs: $G = (V, E)$, $V = \{1, 2, \ldots, n\}$, $b > 0$.
$\forall i, j \in V$, w_{ij} is the weight between i and j.

Definitions: P : set of nodes already in DCST. $P \subseteq V$.
S : set of edges already in DCST. $S \subseteq E$.
$\forall i \in V$, degree$[i]$ is the current degree of i in DCST.
$\forall i \in V$, assume $w_{k'i} = \min_{k \in P} w_{ki}$, let $u[i] = w_{k'i}$, ini$[i] = k'$.

Algorithm:
(1) Let $P = \{1\}$, $S = \emptyset$.
$\forall j \in V$, let ini$[j] = 1$, $u[j] = w_{1j}$.
(2) While $P \neq V$ do
begin
(3) Let $u[k] = \min_{j \in V-P} u[j]$.
(4) $P = P \cup \{k\}$, $S = S \cup \{(\text{ini}[k], k), (k, \text{ini}[k])\}$.
(5) If $P \neq V$ then
$\forall j \in V - P$, if $w_{kj} < u[j]$, let $u[j] = w_{kj}$, ini$[j] = k$.
end
(6) For each $i \in V$ s.t. degree$[i] > b$ do
(7) While degree$[i] > b$ do
begin
(8) For each $j \in V$ do
(9) If $e_{ij} \notin S$ then $f[j] = \infty$
else begin
(10) Let T_i and T_j are the two subtrees reachable from i and j if e_{ij} is removed from S.
(11) If $\neg\exists e \in E$ s.t. $e \neq e_{ij}$ and e connects T_i and T_j then $f[j] = \infty$.
(12) Let $D = \{(x, y) \mid x \in T_i$ and $y \in T_j$ and
{ degree$[x] > b - 1$ and $x \neq i$, or
degree$[j] > b - 1$ and $y \neq j$, or
degree$[j] = 1$ and $j \neq y$, or
$e_{xy} \notin E$, or $e_{xy} \in S$, or $i = x$ }
(13) If $D = T_i \times T_j$ then $f[j] = \infty$
else $f[j] = \max_{(x,y) \in (T_i \times T_j) - D} w_{ij} - w_{xy}$.
end.
(14) If $\forall j \in V, f[j] = \infty$, then error("No feasible solution").
(15) Let $f[k] = \max_{j \in V, f[j] < \infty} f[j]$.
(16) Let x and y be the nodes defining $f[k]$.
(17) Let $S = S - \{e_{ij}\} \cup \{e_{xy}\}$.
end
(18) Return S and its cost.

(Declarative Statement)	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	18	*(Attribute/data-object)*
V	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		the set of nodes
i	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		1, ..., n; nodes of V
ini[]	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		Function; a node corresponding to j
j	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.		a node in V
U[]	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.		an array of weights
W	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.		2-dimensional matrix of weights
P	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.		a subset of nodes
S	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		a subset of edges in a graph
e	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.		an edge
k	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.		a node not in P
b	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.		max. edges connected to a node
E	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.		a set of edges
T	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.		a subset of the nodes in a graph
d[]	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.		degree ofnode
x	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.		a node of Tj
y	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.		a node of Ti
D	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.		a matrix of nodes from Ti and Tj
f[]	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		final matrix
(Conditional Statement)	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	17	*(preCondition)*
	I	.	.	I	.	.	.	.	.	.	.	.	.	.	.	.	.	.		j elem of V
P<>V	I	.	.	.	.	.	I	.	.	.	.	.	.	.	.	.	.	.		nodes in DCST not equal to V
Wkj < U[j]	.	.	.	I	I	I	.	.	.	.	.	.	.	.	.	.	.	.		wt. of current edge < wt.of egde ij
degree[i] > b	.	I	.	.	.	.	.	.	.	.	I	.	.	I	.	.	.	.		degree of i gt. b
	.	I	.	I	.	.	.	I	I	.	.	.	.	.	.	.	.	.		eij not an element of S
	.	I	.	I	.	.	.	.	I	.	.	.	I	.	.	.	.	.		e<>eij and e connects Ti and Tj
	.	I	.	I	.	.	.	.	.	.	.	.	I	.	I	I	.	.		x and y are elements of Ti and Tj resp.
degree[x] > b-1	.	.	.	.	.	.	.	.	.	.	I	.	.	I	I	.	.	.		degree of x gt. > b-1
x<>i	.	I	.	.	.	.	.	.	.	.	.	.	.	.	I	.	.	.		node x not equal node i
degree[j] > b-1	.	.	.	I	.	.	.	.	.	.	I	.	.	I	.	.	.	.		degree of j gt. b-1
y<>j	.	.	.	I	.	.	.	.	.	.	.	.	.	.	.	I	.	.		node y not equal node j
degree[j] = 1	.	.	.	I	.	.	.	.	.	.	.	.	.	I	.	.	.	.		degree of j equals 1
	.	.	.	.	.	.	.	.	I	.	.	I	.	.	I	I	.	.		exy not an element of E
	.	.	.	.	.	.	.	I	I	.	.	.	.	.	I	I	.	.		exy an element of S
i = x	.	I	.	.	.	.	.	.	.	.	.	.	.	.	I	.	.	.		node i equals node x
D = Ti * Tj	.	I	.	I	.	.	.	.	.	.	.	.	I	.	.	.	I	.		D equals product of subtrees Ti & Tj
f[j] = ∞	.	.	.	I	.	.	.	.	.	.	.	.	.	.	.	.	.	I		final wts. not considered any more
(Imperative Statement)	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	17	*(Action)*
P := {1}	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.		let nodes in DCST be 1
S := {}	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		let edges in DCST be 0
ini[j] := 1	.	.	o	I	.	.	.	.	.	.	.	.	.	.	.	.	.	.		let node corresp. to j be 1
U[j] := W1j	.	.	.	I	o	I	.	.	.	.	.	.	.	.	.	.	.	.		assign current wt. to corresp. U[] elem.
U[k] := min(U[j])	.	.	.	I	o	.	.	.	.	I	.	.	.	.	.	.	.	.		assign min of U[i] to U[k]
S := S U{(ini[k],k), (k,ini[k])}	.	.	I	.	.	.	.	b	.	I	.	.	.	.	.	.	.	.		produce new DCST
P := P U {k}	.	.	.	.	.	.	b	.	.	.	.	.	.	.	.	.	.	.		produce new edge subset
U[j] := Wkj	.	.	.	I	o	I	.	.	.	I	.	.	.	.	.	.	.	.		update U[i]
ini[j] := k	.	.	o	I	.	.	.	.	.	I	.	.	.	.	.	.	.	.		update ini[j]
	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		Return S and its cost
f[j] := ∞	.	.	.	I	.	.	.	.	.	.	.	.	.	.	.	.	.	o		do not consider final wts. any more
D := {(x,y)}	.	.	.	.	.	.	.	.	.	.	.	.	.	.	I	I	o	.		update product matrix, D
f[j] = wij -wxy	.	I	.	I	.	I	.	.	.	.	.	.	.	.	I	I	.	o		update final weights
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		error("No feasible solution")
f[k]:=max f[j]	.	.	.	I	.	.	.	.	.	I	.	.	.	.	.	.	.	o		update final weights
	.	.	.	.	.	.	.	.	.	I	.	.	.	.	I	I	.	.		let x & y be nodes defining f[k]
S := S-{eij}U{exy}	.	I	.	I	.	.	.	.	I	.	.	.	.	.	I	I	.	.		update set of edges in DCST
(Conditional Statement)	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	.	*(postCondition)*

Fig. 5.2.3.5 Data flow for Dual algorithm

a) source code: original (see page 145)

1: Let $P = \{1\}$, $S = \emptyset$.
2: $\forall j \in V$, let ini$[j] = 1$, $u[j] = w_{1j}$.
3: While $P \neq V$ do
begin
Let $u[k] = \min_{j \in V-P} u[j]$.
$P = P \cup \{k\}$, $S = S \cup \{(\text{ini}[k], k), (k, \text{ini}[k])\}$.
4: If $P \neq V$ then
5: $\forall j \in V - P$, if $w_{kj} < u[j]$, let $u[j] = w_{kj}$, ini$[j] = k$.
end
6: For each $i \in V$ s.t. degree$[i] > b$ do
7: While degree$[i] > b$ do
begin
8: For each $j \in V$ do
9: If $e_{ij} \notin S$ then $f[j] = \infty$
else begin
10: Let T_i and T_j are the two subtrees reachable from i and j if e_{ij} is removed from S.
11: If $\neg\exists e \in E$ s.t. $e \neq e_{ij}$ and e connects T_i and T_j then $f[j] = \infty$.
12: Let $D = \{(x, y) \mid x \in T_i$ and $y \in T_j$ and
$\{$ degree$[x] > b - 1$ and $x \neq i$, or
degree$[j] > b - 1$ and $y \neq j$, or
degree$[j] = 1$ and $j \neq y$, or
$e_{xy} \notin E$, or $e_{xy} \in S$, or $i = x$ $\}$
13: If $D = T_i \times T_j$ then $f[j] = \infty$
else $f[j] = \max_{(x,y) \in (T_i \times T_j) - D} w_{ij} - w_{xy}$.
end.
14: If $\forall j \in V, f[j] = \infty$, then error("No feasible solution").
Let $f[k] = \max_{j \in V, f[j] < \infty} f[j]$.
Let x and y be the nodes defining $f[k]$.
Let $S := S - \{e_{ij}\} \cup \{e_{xy}\}$.
end
15: Return S and its cost.

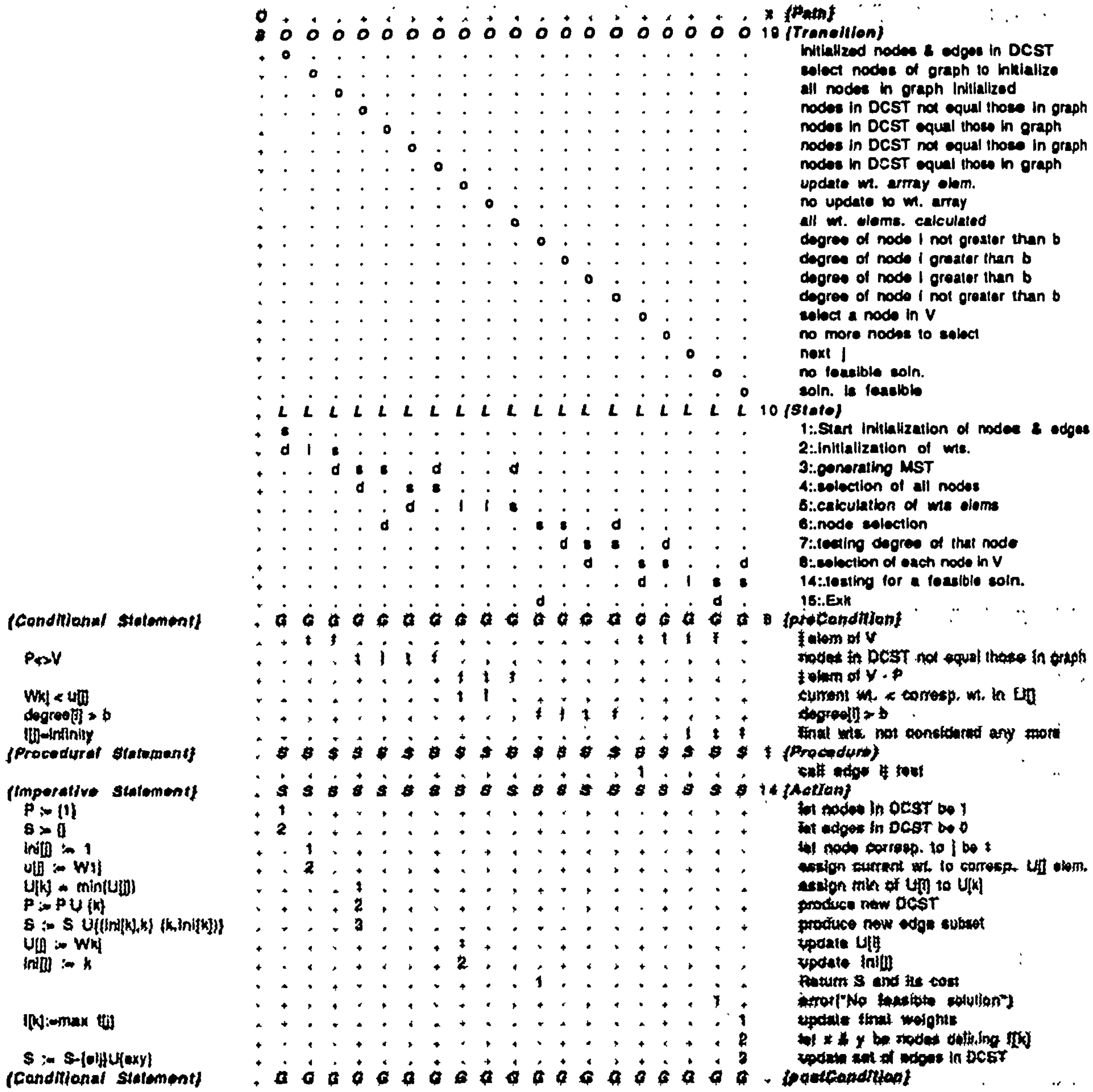
b) edited source code: line nos removed, states 1..15 identified

	O	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	19	*(Transition)*
	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		
	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		
	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		
	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		
	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.		
	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.		
	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.		
	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.		
	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		
	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.		
	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.		
	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.		
	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.		
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.		
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.		
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.		
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.		
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.		
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		
	.	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	10	*(State)*
	.	s	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		1.
	.	d	l	s	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		2.
	.	.	.	d	s	s	.	d	.	.	d	.	.	.	.	.	.	.	.	.		3.
	.	.	.	.	d	.	s	s	.	.	.	.	.	.	.	.	.	.	.	.		4.
	.	.	.	.	.	.	d	.	l	l	s	.	.	.	.	.	.	.	.	.		5.
	.	.	.	.	.	d	.	.	.	.	.	s	s	.	d	.	.	.	.	.		6.
	.	.	.	.	.	.	.	.	.	.	.	.	d	s	s	.	d	.	.	.		7.
	.	.	.	.	.	.	.	.	.	.	.	.	.	d	.	s	s	.	.	d		8.
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	d	.	l	s	s		14.
	.	.	.	.	.	.	.	.	.	.	.	d	.	.	.	.	.	.	d	.		15.
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	6	*(preCondition)*
	.	.	t	f	.	.	.	.	.	.	.	.	.	.	.	t	f	t	f	.		j elem of V
P<>V	.	.	.	.	t	f	t	f	.	.	.	.	.	.	.	.	.	.	.	.		nodes in DCST not equal those in graph
	.	.	.	.	.	.	.	.	t	t	f	.	.	.	.	.	.	.	.	.		j elem of V - P
Wkj < u[j]	.	.	.	.	.	.	.	.	t	f	.	.	.	.	.	.	.	.	.	.		current wt. < corresp. wt. in U[]
degree[i] > b	.	.	.	.	.	.	.	.	.	.	.	f	t	t	f	.	.	.	.	.		degree[i] > b
f[j]=infinity	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	t	f		final wts. not considered any more
(Procedural Statement)	.	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	1	*(Procedure)*
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.		call edge ij test
(Imperative Statement)	.	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	14	*(Action)*
P := {1}	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		let nodes in DCST be 1
S := {}	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		let edges in DCST be 0
ini[j] := 1	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		let node corresp. to j be 1
u[j] := W1j	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		assign current wt. to corresp. U[] elem.
U[k] = min(U[j])	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		assign min of U[i] to U[k]
P := P U {k}	.	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		produce new DCST
S := S U{(ini[k],k) (k,ini[k])}	.	.	.	.	3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		produce new edge subset
U[j] := Wkj	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.		update U[i]
ini[j] := k	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.		update ini[j]
	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.		Return S and its cost
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.		error("No feasible solution")
f[k]:=max f[j]	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1		update final weights
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2		let x & y be nodes defining f[k]
S := S-{eij}U{exy}	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	3		update set of edges in DCST
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	.	*(postCondition)*

c) Normalised infoMap: shaded areas to be filled in

	O	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	*{Path}*
	s	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	19	*{Transition}*
		o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		initialized nodes & edges in DCST
	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		select nodes of graph to initialize
	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		all nodes in graph initialized
	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		nodes in DCST not equal those in graph
	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.		nodes in DCST equal those in graph
	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.		nodes in DCST not equal those in graph
	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.		nodes in DCST equal those in graph
	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.		update wt. array elem.
	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		no update to wt. array
	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.		all wt. elems. calculated
	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.		degree of node i not greater than b
	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.		degree of node i greater than b
	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.		degree of node i greater than b
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.		degree of node i not greater than b
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.		select a node in V
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.		no more nodes to select
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.		next j
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.		no feasible soln.
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		soln. is feasible
	.	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	10	*{State}*
	.	s	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		1:Start initialization of nodes & edges
	.	d	l	s	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		2:initialization of wts.
	.	.	.	d	s	s	.	d	.	.	d	.	.	.	.	.	.	.	.	.		3:generating MST
	.	.	.	.	d	.	s	s	.	.	.	.	.	.	.	.	.	.	.	.		4:selection of all nodes
	.	.	.	.	.	.	d	.	l	l	s	.	.	.	.	.	.	.	.	.		5:calculation of wts elems
	.	.	.	.	.	d	.	.	.	.	.	s	s	.	d	.	.	.	.	.		6:node selection
	.	.	.	.	.	.	.	.	.	.	.	.	d	s	s	.	d	.	.	.		7:testing degree of that node
	.	.	.	.	.	.	.	.	.	.	.	.	.	d	.	s	s	.	.	d		8:selection of each node in V
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	d	.	l	s	s		14:testing for a feasible soln.
	.	.	.	.	.	.	.	.	.	.	.	d	.	.	.	.	.	.	d	.		15:Exit
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	6	*{preCondition}*
	.	.	t	f	.	.	.	.	.	.	.	.	.	.	.	t	f	t	f	.		j elem of V
P<>V	.	.	.	.	t	f	t	f	.	.	.	.	.	.	.	.	.	.	.	.		nodes in DCST not equal those in graph
	.	.	.	.	.	.	.	.	t	t	f	.	.	.	.	.	.	.	.	.		j elem of V - P
Wkj < u[j]	.	.	.	.	.	.	.	.	t	f	.	.	.	.	.	.	.	.	.	.		current wt. < corresp. wt. in U[]
degree[i] > b	.	.	.	.	.	.	.	.	.	.	.	f	t	t	f	.	.	.	.	.		degree[i] > b
f[j]=infinity	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	t	f		final wts. not considered any more
{Procedural Statement}	.	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	1	*{Procedure}*
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.		call edge ij test
{Imperative Statement}	.	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	14	*{Action}*
P := {1}	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		let nodes in DCST be 1
S := {}	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		let edges in DCST be 0
ini[j] := 1	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		let node corresp. to j be 1
u[j] := W1j	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		assign current wt. to corresp. U[] elem.
U[k] = min(U[j])	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		assign min of U[j] to U[k]
P := P U {k}	.	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		produce new DCST
S := S U{(ini[k],k) (k,ini[k])}	.	.	.	.	3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		produce new edge subset
U[j] := Wkj	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.		update U[j]
ini[j] := k	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.		update ini[j]
	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.		Return S and its cost
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.		error("No feasible solution")
f[k]:=max f[j]	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1		update final weights
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2		let x & y be nodes defining f(k)
S := S-{eij}U{exy}	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	3		update set of edges in DCST
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	.	*{postCondition}*

d) Normalised InfoMap: states and transitions added (shaded area)



e) Normalised infoMap: state diagram (unshaded area)

	O	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	19	*(Transition)*
	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		initialized nodes & edges in DCST
	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		select nodes of graph to initialize
	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		all nodes in graph initialized
	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		nodes in DCST not equal those in graph
	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.		nodes in DCST equal those in graph
	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.		nodes in DCST not equal those in graph
	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.		nodes in DCST equal those in graph
	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.		*update wt. array elem.*
	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		no update to wt array
	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.		*all wt. elems. calculated*
	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.		degree of node i not greater than b
	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.		degree of node i greater than b
	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.		degree of node i greater than b
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.		degree of node i not greater than b
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.		select a node in V
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.		no more nodes to select
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.		next j
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.		no feasible soln
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		soln. is feasible
	.	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	10	*(State)*
	.	s	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		1:.Start initialization of nodes & edges
	.	d	l	s	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		2:.initialization of wts.
	.	.	.	d	s	s	.	d	.	.	d	.	.	.	.	.	.	.	.	.		3:.generating MST
	.	.	.	.	d	.	s	s	.	.	.	.	.	.	.	.	.	.	.	.		4:.selection of all nodes
	.	.	.	.	.	.	d	.	f	l	s	.	.	.	.	.	.	.	.	.		5:.*calculation of wts elems*
	.	.	.	.	.	d	.	.	.	.	.	s	s	.	d	.	.	.	.	.		6:.node selection
	.	.	.	.	.	.	.	.	.	.	.	.	d	s	s	.	d	.	.	.		7:.*testing degree of that node*
	.	.	.	.	.	.	.	.	.	.	.	.	.	d	.	s	s	.	.	d		8..selection of each node in V
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	d	.	f	s	s		14:.testing for a feasible soln.
	.	.	.	.	.	.	.	.	.	.	.	d	.	.	.	.	.	.	d	.		15:.Exit
(Conditional Statement)	.	G	G	G	C	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	6	*(preCondition)*
	.	.	t	'	.	.	.	.	.	.	.	.	.	.	.	t	t	t	f	.		j elem of V
P<>V	.	.	.	.	t	f	t	f	.	.	.	.	.	.	.	.	.	.	.	.		nodes in DCST not equal those in graph
	.	.	.	.	.	.	.	.	f	t	f	.	.	.	.	.	.	.	.	.		j elem of V - P
Wkj < u[j]	.	.	.	.	.	.	.	.	t	l	.	.	.	.	.	.	.	.	.	.		current wt. < corresp wt. in U[]
degree[i] > b	.	.	.	.	.	.	.	.	.	.	.	f	f	t	f	.	.	.	.	.		degree[i] > b
f[j]=infinity	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	t	f		final wts not considered any more
(Procedural Statement)	.	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	1	*(Procedure)*
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.		call edge ij test
(Imperative Statement)	.	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	14	*(Action)*
P := {1}	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		let nodes in DCST be 1
S := {}	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		let edges in DCST be 0
ini[j] := 1	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		let node corresp. to j be 1
u[j] := W1j	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		assign current wt. to *corresp. U[] elem.*
U[k] = min(U[j])	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		assign min of U[i] to U[k]
P := P U {k}	.	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		produce new DCST
S := S U{(ini[k],k) (k,ini[k])}	.	.	.	.	3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		produce new edge subset
U[j] := Wkj	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.		update U[i]
ini[j] := k	.	.	.	.	.	.	.	.	2	.	.	.	.	.	.	.	.	.	.	.		update ini[j]
	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.		Return S and its cost
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.		error("No feasible solution")
f[k]:=max f[j]	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1		update final weights
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2		let x & y be nodes defining f[k]
S := S-{eij}U{exy}	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	3		update set of edges in DCST
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	.	*(postCondition)*

f) Normalized InfoMap to be optimised: shaded area

```
                                      O . . . . . . . . . . . . . . . .  x  {Path}
                                      s O O O O O O O O O O O O O O O O  16 {Transition}
                                      . o . . . . . . . . . . . . . . .     select nodes of graph to initialize
                                      . . o . . . . . . . . . . . . . .     all nodes in graph initialized
                                      . . . o . . . . . . . . . . . . .     nodes in DCST not equal those in graph
                                      . . . . o . . . . . . . . . . . .     nodes in DCST equal those in graph
                                      . . . . . o . . . . . . . . . . .     update wt. array elem.
                                      . . . . . . o . . . . . . . . . .     no update to wt. array
                                      . . . . . . . o . . . . . . . . .     all wt. elems. calculated
                                      . . . . . . . . o . . . . . . . .     nodes in DCST equal those in graph
                                      . . . . . . . . . o . . . . . . .     no more nodes to search for
                                      . . . . . . . . . . o . . . . . .     node found
                                      . . . . . . . . . . . o . . . . .     continue search for node
                                      . . . . . . . . . . . . o . . . .     select a node in V
                                      . . . . . . . . . . . . . o . . .     no more nodes to select
                                      . . . . . . . . . . . . . . o . .     next j
                                      . . . . . . . . . . . . . . . o .     no feasible soln.
                                      . . . . . . . . . . . . . . . . o     soln. is feasible
                                      . L L L L L L L L L L L L L L L L  7  {State}
                                      . l s . . . . . . . . . . . . . .     1:.Start initializing nodes, edges, wts.
                                      . . d s s . . d d . . . . . . . .     2:.generating MST
                                      . . . d . l l s s . . . . . . . .     3:.calculation of wts elems
                                      . . . . d . . . . s s l . d . . .     4:.search for node i with di > b
                                      . . . . . . . . . . d . s s . . d     5:. selecting a corresponding node j
                                      . . . . . . . . . . . . d . l s s     14:.testing for a feasible soln.
                                      . . . . . . . . . d . . . . . d .     15:.Exit
{Conditional Statement}               . G G G G G G G G G G G G G G G G  6  {preCondition}
                                      . t f . . . . . . f t t t f t f .     j elem of V
  P<>V                                . . . t f t t t f . . . . . . . .     nodes in DCST not equal those in graph
                                      . . . . . t t f . . . . . . . . .     j elem of V - P
  Wkj < U[j]                          . . . . . t f . . . . . . . . . .     current wt. < corresp. wt. in U[]
  degree[i] > b                       . . . . . . . . . t f . . . . . .     degree[i] > b
  f[j]==                              . . . . . . . . . . . . . . t t f     final wts. not considered any more
{Procedural Statement}                . S S S S S S S S S S S S S S S S  1  {Procedure}
                                      . . . . . . . . . . . 1 . . . . .     call edge ij test
{Imperative Statement}                . S S S S S S S S S S S S S S S S  14 {Action}
  Ini[j] := 1                         . 1 . . . . . . . . . . . . . . .     let node corresp. to j be 1
  U[j] := W1j                         . 2 . . . . . . . . . . . . . . .     assign current wt. to corresp. U[] elem.
  P := {1}                            . . 1 . . . . . . . . . . . . . .     let nodes in DCST be 1
  S := {}                             . . 2 . . . . . . . . . . . . . .     let edges in DCST be 0
  U[k] = min(U[j])                    . . . 1 . . . . . . . . . . . . .     assign min of U[i] to U[k]
  P := P U {k}                        . . . 2 . . . . . . . . . . . . .     produce new DCST
  S := S U{(Ini[k],k) (k,Ini[k])}     . . . 3 . . . . . . . . . . . . .     produce new edge subset
  U[j] := Wkj                         . . . . . 1 . . . . . . . . . . .     update U[i]
  Ini[j] := k                         . . . . . 2 . . . . . . . . . . .     update Ini[j]
                                      . . . . . . . . . 1 . . . . . . .     Return S and its cost
                                      . . . . . . . . . . . . . . . 1 .     error("No feasible solution")
  f[k]:=max f[j]                      . . . . . . . . . . . . . . . . 1     update final weights
                                      . . . . . . . . . . . . . . . . 2     let x & y be nodes defining f[k]
  S := S-{eij}U{exy}                  . . . . . . . . . . . . . . . . 3     update set of edges in DCST
{Conditional Statement}               . G G G G G G G G G G G G G G G G  .  {postCondition}
```

g) Optimized and documented infoMap

Note: Steps - Optimized state diagram as Structured English (generated) and source code (generated) - are to be done as in Fig. 5.2.4.1

Fig. 5.2.4.3 Control flow for Dual algorithm (main)

	O	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	*{Path}*
	S	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	15	*{Transition}*
	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.		
	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.		
	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.		
	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.		
	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		
	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.		
	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.		
	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.		
	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.		
	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.		
	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.		
	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.		
	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.		
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.		
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		
	.	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	6	*{State}*
	.	s	s	.	.	.	.	.	.	.	.	.	.	.	.	.		9
	.	.	d	s	s	.	.	.	.	.	.	.	.	.	.	.		10
	.	.	.	d	d	s	s	.	.	.	.	.	.	.	.	.		11
	.	.	.	.	.	d	d	s	s	s	s	s	s	s	.	.		12
	.	.	.	.		.	.	d	d	d	d	d	d	d	s	s		13
	.	d	.	.	.	.	.	.	.	.	.	.	.	.	d	d		14
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	14	*{preCondition}*
	.	t	f	.	.	.	.	.	.	.	.	.	.	.	.	.		eij not an element of S
	.	.	.	t	f	.	.	.	.	.	.	.	.	.	.	.		*eij removed from S*
e<>eij	.	.	.	.	.	t	c	.	.	.	.	.	.	.	.	.		edge not equal edge ij
	.	.	.	.	.	t	c	.	.	.	.	.	.	.	.	.		e connects Ti and Tj
	.	.	.	.	.	.	.	t	t	t	t	t	t	c	.	.		x & y elems of Ti and Tj resp.
degree[x]> b-1	.	.	.	.	.	.	.	t	.	.	.	.	.	c	.	.		degree of x gt. b-1
x <> i	.	.	.	.	.	.	.	t	.	.	.	.	.	c	.	.		*node x not equal node i*
degree[j]> b-1	.	.	.	.	.	.	.	.	t	.	.	.	.	c	.	.		degree of y gt. b-1
y <> j	.	.	.	.	.	.	.	.	t	.	.	.	.	c	.	.		node y not equal node j
degree[j] = 1	.	.	.	.	.	.	.	.	.	t	.	.	.	c	.	.		degree of j equals 1
	.	.	.	.	.	.	.	.	.	.	t	.	.	c	.	.		exy not an element of E
	.	.	.	.	.	.	.	.	.	.	.	t	.	c	.	.		*exy an element of S*
i = x	.	.	.	.	.	.	.	.	.	.	.	.	t	c	.	.		node i equals node x
D = Ti * Tj	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	f		D equals product of subtrees Ti & Tj
{Imperative Statement}	.	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	4	*{Action}*
f[j] := ∞	.	1	.	.	.	1	.	.	.	.	.	.	.	.	1	.		do not consider wts. any more
	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.		*let Ti & Tj be subtrees*
D := f(x y)	.	.	.	.	.	.	.	1	1	1	1	1	1	.	.	.		update product matrix, D
f[j]:= wij - wxy	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1		update final weights
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	.	*{postCondition}*

c) Normalized infoMap: shaded areas to be filled in

	O	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	*{Path}*
	S	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	15	*{Transition}*
	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.		edge ij not in DCST
	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.		edge ij already in DCST
	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.		edge ij removed from S
	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.		edge ij is in S
	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		an edge not equal to edge ij is found
	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.		found another edge
	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.		node x different from node i
	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.		node y different from node j
	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.		degree of j equals 1
	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.		exy not an element of E
	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.		exy an element already in DCST
	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.		node i equals node x
	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.		no update to D
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.		wt. not considered any more
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		update final weights
	.	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	6	*{State}*
	.	s	s	.	.	.	.	.	.	.	.	.	.	.	.	.		9: Start evaluating edge ij
	.	.	d	s	s	.	.	.	.	.	.	.	.	.	.	.		10: creating subtrees
	.	.	.	d	d	s	s	.	.	.	.	.	.	.	.	.		11: finding another edge
	.	.	.	.	.	d	d	s	s	s	s	s	s	s	.	.		12: calculating D
	.	.	.	.	.	.	.	d	d	d	d	d	d	d	s	s		13: evaluating D
	.	d	.	.	.	.	.	.	.	.	.	.	.	.	d	d		14: testing for a feasible solution
{Conditional Statement}	.	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	14	*{preCondition}*
	.	t	f	.	.	.	.	.	.	.	.	.	.	.	.	.		eij not an element of S
	.	.	.	t	f	.	.	.	.	.	.	.	.	.	.	.		eij removed from S
e<>eij	.	.	.	.	.	t	c	.	.	.	.	.	.	.	.	.		edge not equal edge ij
	.	.	.	.	.	t	c	.	.	.	.	.	.	.	.	.		e connects Ti and Tj
	.	.	.	.	.	.	.	t	t	t	t	t	t	c	.	.		x & y elems of Ti and Tj resp.
degree[x]> b-1	.	.	.	.	.	.	.	t	.	.	.	.	.	c	.	.		degree of x gt. b-1
x <> i	.	.	.	.	.	.	.	t		.	.	.	.	c	.	.		node x not equal node i
degree[j]> b-1	.	.	.	.	.	.	.	.	t	.	.	.	.	c	.	.		degree of y gt. b-1
y <> j	.	.	.	.	.	.	.	.	t	.	.	.	.	c	.	.		node y not equal node j
degree[j] = 1	.	.	.	.	.	.	.	.	.	t	.	.	.	c	.	.		degree of j equals 1
	.	.	.	.	.	.	.	.	.	.	t	.	.	c	.	.		exy not an element of E
	.	.	.	.	.	.	.	.	.	.	.	t	.	c	.	.		exy an element of S
i = x	.	.	.	.	.	.	.	.	.	.	.	.	t	c	.	.		node i equals node x
D = Ti * Tj	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	f		D equals product of subtrees Ti & Tj
{Imperative Statement}	.	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	4	*{Action}*
f[j] := ∞	.	1	.	.	.	1	.		.	.	.	.	.	.	1	.		do not consider wts. any more
	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.		let Ti & Tj be subtrees
D := f(x y)	.	.	.	.	.	.	.	1	1	1	1	1	1	.	.	.		update product matrix, D
f[j]:= wij - wxy	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1		update final weights
{Conditional Statement}	.	*G*	*C*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	*G*	.	*{postCondition}*

d) Normalized infoMap: states and transitions added (shaded area)

	Q	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	{Path}
	S	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	15	{Transition}
	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.		edge ij not in DCST
	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.		edge ij already in DCST
	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.		edge ij removed from S
	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.		edge ij is in S
	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		an edge not equal to edge ij is found
	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.		found another edge
	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.		node x different from node i
	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.		node y different from node j
	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.		degree of j equals 1
	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.		exy not an element of E
	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.		exy an element already in DCST
	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.		node i equals node x
	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.		no update to D
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.		wt. not considered any more
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		update final weights
	.	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	6	{State}
	.	s	s	.	.	.	.	.	.	.	.	.	.	.	.	.		9:.Start evaluating edge ij
	.	.	d	s	s	.	.	.	.	.	.	.	.	.	.	.		10:.creating subtrees
	.	.	.	d	d	s	s	.	.	.	.	.	.	.	.	.		11:.finding another edge
	.	.	.	.	.	d	d	s	s	s	s	s	s	s	.	.		12:.calculating D
	.	.	.	.	.	.	.	d	d	d	d	d	d	d	s	s		13:.evaluating D
	.	d	.	.	.	.	.	.	.	.	.	.	.	.	d	d		14:.testing for a feasible solution
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	14	{preCondition}
	.	t	t	.	.	.	.	.	.	.	.	.	.	.	.	.		eij not an element of S
	.	.	.	t	t	.	.	.	.	.	.	.	.	.	.	.		eij removed from S
a<>eij	.	.	.	.	.	t	c	.	.	.	.	.	.	.	.	.		edge not equal edge ij
	.	.	.	.	.	t	c	.	.	.	.	.	.	.	.	.		e connects Ti and Tj
	.	.	.	.	.	.	.	t	t	t	t	t	t	c	.	.		x & y elems of Ti and Tj resp.
degree[x]> b-1	.	.	.	.	.	.	.	t	.	.	.	.	.	c	.	.		degree of x gt. b-1
x <> i	.	.	.	.	.	.	.	t	.	.	.	.	.	c	.	.		node x not equal node i
degree[j]> b-1	.	.	.	.	.	.	.	.	t	.	.	.	.	c	.	.		degree of y gt. b-1
y <> j	.	.	.	.	.	.	.	.	t	.	.	.	.	c				node y not equal node j
degree[j] = 1	.	.	.	.	.	.	.	.	.	t	.	.	.	c	.	.		degree of j equals 1
	.	.	.	.	.	.	.	.	.	.	t	.	.	c	.	.		exy not an element of E
	.	.	.	.	.	.	.	.	.	.	.	t	.	c	.	.		exy an element of S
i = x	.	.	.	.	.	.	.	.	.	.	.	.	t	c	.	.		node i equals node x
D = Ti * Tj	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	f		D equals product of subtrees Ti & Tj
{Imperative Statement}	.	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	4	{Action}
t[ij] := ∞	.	1	.	.	.	1	.	.	.	.	.	.	.	.	1	.		do not consider wts. any more
	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.		let Ti & Tj be subtrees
D := f(x y)	.	.	.	.	.	.	.	1	1	1	1	1	1	.	.	.		update product matrix, D
t[ij]:= wij - wxy	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1		update final weights
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	.	{postCondition}

e) Normalized infoMap: state diagram (unshaded area)

	O	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	*{Path}*
	S	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	15	*{Transition}*
	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.		edge ij not in DCST
	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.		edge ij already in DCST
	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.		edge ij removed from S
	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.		edge ij is in S
	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		an edge not equal to edge ij is found
	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.		found another edge
	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.		node x different from node i
	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.		node y different from node j
	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.		degree of j equals 1
	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.		exy not an element of E
	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.		exy an element already in DCST
	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.		node i equals node x
	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.		no update to D
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.		wt. not considered any more
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		update final weights
	.	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	6	*{State}*
	.	s	s	.	.	.	.	.	.	.	.	.	.	.	.	.		9:.Start evaluating edge ij
	.	.	d	s	s	.	.	.	.	.	.	.	.	.	.	.		10:.creating subtrees
	.	.	.	d	d	s	s	.	.	.	.	.	.	.	.	.		11:.finding another edge
	.	.	.	.	.	d	d	s	s	s	s	s	s	s	.	.		12:.calculating D
	.	.	.	.	.	.	.	d	d	d	d	d	d	d	s	s		13:.evaluating D
	.	d	.	.	.	.	.	.	.	.	.	.	.	.	d	d		14:.testing for a feasible solution
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	14	*{preCondition}*
	.	t	f	.	.	.	.	.	.	.	.	.	.	.	.	.		eij not an element of S
	.	.	.	t	t	.	.	.	.	.	.	.	.	.	.	.		eij removed from S
e<>eij	.	.	.	.	.	t	c	.	.	.	.	.	.	.	.	.		edge not equal edge ij
	.	.	.	.	.	t	c	.	.	.	.	.	.	.	.	.		e connects Ti and Tj
	.	.	.	.	.	.	.	t	t	t	t	t	t	c	.	.		x & y elems of Ti and Tj resp.
degree[x]> b-1	.	.	.	.	.	.	.	t	.	.	.	.	.	c	.	.		degree of x gt. b-1
x <> i	.	.	.	.	.	.	.	t	.	.	.	.	.	c	.	.		nodo x not equal node i
degree[j]> b-1	.	.	.	.	.	.	.	.	t	.	.	.	.	c	.	.		degree of y gt. b-1
y <> j	.	.	.	.	.	.	.	.	t	.	.	.	.	c	.	.		node y not equal node j
degree[j] = 1	.	.	.	.	.	.	.	.	.	t	.	.	.	c	.	.		degree of j equals 1
	.	.	.	.	.	.	.	.	.	.	t	.	.	c	.	.		exy not an element of E
	.	.	.	.	.	.	.	.	.	.	.	t	.	c	.	.		exy an element of S
i = x	.	.	.	.	.	.	.	.	.	.	.	.	t	c	.	.		node i equals node x
D = Ti * Tj				.	.	.	.	.	.	.	.	.	.	.	t	f		D equals product of subtrees Ti & Tj
{Imperative Statement}	.	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	4	*{Action}*
f[j] := ∞		1	.	.	.	1	.	.	.	.	.	.	.	.	1	.		do not consider wts. any more
	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.		let Ti & Tj be subtrees
D := f(x y)	.	.	.	.	.	.	.	1	1	1	1	1	1	.	.	.		update product matrix, D
f[j] = wij - wxy	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1		update final weights
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	.	*{postCondition}*

f) Normalized infoMap to be optimized: shaded area

	O	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	*{Path}*
	S	O	O	O	O	O	O	O	O	O	O	O	O	O	O	14	*{Transition}*
	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.		edge ij not in DCST
	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.		edge ij already in DCST
	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.		edge ij removed from S
	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		found an edge not eq. to edge ij
	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.		found another edge
	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.		node x different from node i
	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.		node y different from node j
	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.		degree of j equals 1
	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.		exy not an element of E
	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.		exy an element already in DCST
	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.		node i equals node x
	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.		no update to D
	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.		wt. not considered any more
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		update final weights
	.	L	L	L	L	L	L	L	L	L	L	L	L	L	L	5	*{State}*
	.	s	s	.	.	.	.	.	.	.	.	.	.	.	.		9:.Start evaluating edge ij
	.	.	d	s	s	s	.	.	.	.	.	.	.	.	.		10:.creating subtrees
	.	.	.	d	d	d	s	s	s	s	s	s	s	.	.		12:.calculating D
	.	.	.	.	.	.	d	d	d	d	d	d	d	s	s		13:.evaluating D
	.	d	.	.	.	.	.	.	.	.	.	.	.	d	d		14:.testing for a feasible solution
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	14	*{preCondition}*
	.	t	f	.	.	.	.	.	.	.	.	.	.	.	.		eij not an element of S
	.	.	.	t	f	f	.	.	.	.	.	.	.	.	.		eij removed from S
e<>eij	.	.	.	.	t	c	.	.	.	.	.	.	.	.	.		edge not equal edge ij
	.	.	.	.	t	c	.	.	.	.	.	.	.	.	.		e connects Ti and Tj
	.	.	.	.	.	.	t	t	t	t	t	t	c	.	.		x & y elems of Ti and Tj resp.
degree[x]> b-1	.	.	.	.	.	.	t	.	.	.	.	.	c	.	.		degree of x gt. b-1
x<>i	.	.	.	.	.	.	t	.	.	.	.	.	c	.	.		node x not equal node i
degree[j]> b-1	.	.	.	.	.	.	.	t	.	.	.	.	c	.	.		degree of y gt b-1
y<>j	.	.	.	.	.	.	.	t	.	.	.	.	c	.	.		node y not equal node j
degree[j] = 1	.	.	.	.	.	.	.	.	t	.	.	.	c	.	.		degree of j equals 1
	.	.	.	.	.	.	.	.	.	t	.	.	c	.	.		exy not an element of E
	.	.	.	.	.	.	.	.	.	.	t	.	c		.		exy an element of S
i = x	.	.	.	.	.	.	.	.	.	.	.	t	c	.	.		node i equals node x
D = Ti * Tj	.	.	.	.	.	.	.	.	.	.	.	.	.	t	f		D equals product of subtrees Ti & Tj
{Imperative Statement}	.	S	S	S	S	S	S	S	S	S	S	S	S	S	S	4	*{Action}*
f[j] := ∞	.	1	.	.	1	.	.	.	.	.	.	.	.	1	.		do not consider wts. any more
	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.		let Ti & Tj be subtrees
D := [(x y)	.	.	.	.	.	.	1	1	1	1	1	1	.	.	.		update product matrix, D
f[j]:=max wij -wxy	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1		update final weights
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	.	*{postCondition}*

g) Optimized and documented infoMap

Note: Steps - Optimized state diagram as Structured English (generated) and source code (generated) - are to be done as in Fig 5.2.4.1

Fig. 5.2.4.4 Control flow for Dual algorithm (procedure)

Anneal algorithm:

Inputs: $G = (V, E)$, $V = \{1, 2, \ldots, n\}$, $b > 0$.
$\forall e \in E$, $w[e]$ is the weight for edge e.
Parameters for cooling schedule: M, T_0, and T_f. $\beta = (t_0 - t_f)/(M t_0 t_f)$.

Definitions: S : set of edges already in DCST. $S \subseteq E$.
$\forall i \in V$, degree[i] is the current degree of i in DCST.

Algorithm:

(1) Use gen_dcst() to generate S for a DCST.
(2) Initialize queues Q_1 and Q_2 to be empty. Let $t_i = t_0$.
(3) For all $e \in S$, enqueue(Q_1, e); for all $e \in E - S$, enqueue(Q_2, e).
(4) While $t_i > t_f$ do
begin
(5) dequeue(Q_1, e_{ij}).
(6) If degree[i] $= 1$ or degree[j] $= 1$ then
enqueue(Q_1, e_{ij}), let $t_i = t_i/(1 + \beta t_i)$, goto (5).
(7) Loop $|E - S|$ times
begin
(8) dequeue(Q_2, e_{xy}).
(9) If degree[x] $= b$ or degree[y] $= b$ then enqueue(Q_2, e_{xy}), goto (17).
(10) If e_{xy} cannot connect the two subtrees created by removing
e_{ij} from S, then enqueue(Q_2, e_{xy}), goto (17).
(11) Let $\Delta = w_{xy} - w_{ij}$, r be a random number in $[0, 1]$.
(12) If $\Delta \leq 0$ or $e^{-\Delta/t_i} > r$ then
begin
(13) Let $S = S - \{e_{ij}\} \cup \{e_{xy}\}$.
(14) enqueue(Q_1, e_{xy}), enqueue(Q_2, e_{ij}).
(15) Let $t_i = t_i/(1 + \beta t_i)$.
end
(16) enqueue(Q_2, e_{xy}), let $t_i = t_i/(1 + \beta t_i)$.
(17) end {loop}
end
(18) Return S and its cost for the best solution encountered.

(Declarative Statement)	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	18	*(Attribute/DataObject)*
i	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		1, ..., n; nodes of V
j	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		a node in V
W	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		2-dimensional matrix of weights
S	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.		a subset of edges in a graph
e	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.		an edge
cost of S	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.		summation of weights in a tree
E	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.		a set of edges
x	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.		a node of Tj
y	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.		a node of Ti
degree[]	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.		degree array
DCST	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.		Degree-constrained Spanning Tree
r	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.		a random number between 0 nad 1
ti	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.		a temperature
tf	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.		a reducer
β	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.		beta, a constant use in temp. calculation
Δ	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.		a difference between two weights
Q1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.		queue with edges already in S
Q2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		queue with edge not yet in S
(Conditional Statement)	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	9	*(preCondition)*
ti>tf	i	i	.	.	.	.	.	.	.	.	.	.	i	i	.	.	.	.		initial temp gt. final temp
degree[i] = 1	i	.	.	.	.	.	.	.	.	i	.	.	.	.	.	.	.	.		degree of i equals1
degree[j] = 1	.	i	.	.	.	.	.	.	.	i	.	.	.	.	.	.	.	.		degree of j equals 1
Loop |E-S| times	.	.	.	i	.	.	i	.	.	.	.	.	.	.	.	.	.	.		Loop <= mod of E-S times
degree[x] = b	.	.	.	.	i	.	.	i	.	i	.	.	.	.	.	.	.	.		degree of x equals b
degree[y] = b	.	.	.	.	.	.	.	.	i	i	.	.	.	.	.	.	.	.		degree of y equals b
	.	.	.	.	.	.	.	i	i	i	.	.	.	.	.	.	.	.		exy cannot connect the two subtrees
Δ <= 0	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	i	.	.		difference in wts. less than or equals zero
(e**-(Δ/ti))>r	i	.	.	.	i	.	.	.	.	.	.	i	i	.	.	i	.	.		number gt. 1
(Procedural Statement)	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	1	*(Procedure)*
	.	.	.	i	.	.	.	.	.	.	i	.	.	.	.	.	.	.		Use gen_dcst to generate S
(Imperative Statement)	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	16	*(Action)*
Q1:= {}	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.		let Queue 1 be empty
Q2:= {}	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		let Queue 2 be empty
ti := t0	i	.	.	.	.	.	.	.	.	.	.	.	b	.	.	.	.	.		let initial temperature be t0
enqueue(Q1,e)	.	.	.	.	i	.	.	.	.	.	.	.	.	.	.	.	b	.		enqueue edge,e to Queue 1
enqueue(Q2,e)	.	.	.	.	i	.	.	.	.	.	.	.	.	.	.	.	.	b		enqueue edge,e to Queue 2
	.	.	.	o	.	o	.	.	.	.	.	.	.	.	.	.	.	.		Return S and its optimum cost
dequeue(Q1,eij)	i	i	.	.	i	.	.	.	.	.	.	.	.	.	.	.	b	.		remove edge from Queue 1
enqueue(Q1,eij)	i	i	.	.	i	.	.	.	.	.	.	.	.	.	.	.	b	.		enqueue edge to Queue 1
ti:= ti/(1+ beta_ti)	i	.	.	.	.	.	.	.	.	.	.	.	b	.	i	.	.	.		calculate new ti
dequeue(Q2,exy)	.	.	.	.	i	.	.	i	i	.	.	.	.	.	.	.	.	b		remove edge from Queue 2
enqueue(Q2,exy)	.	.	.	.	i	.	.	i	i	.	.	.	.	.	.	.	.	b		enqueue edge to Queue 2
Δ:= wxy-wij	i	i	i	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.		calculate wt. difference
	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.		r be a random number in [0,1]
S:= S-{eij}U{exy}	i	i	.	b	i	.	.	i	i	.	.	.	.	.	.	.	.	.		update set of edges in DCST
enqueue(Q1,exy)	.	.	.	.	i	.	.	i	i	.	.	.	.	.	.	.	b	.		enqueue edge xy to Queue 1
enqueue(Q2,eij)	i	i	.	.	i	.	.	.	.	.	.	.	.	.	.	.	.	b		enqueue edge ij to Queue 2
(Conditional Statement)	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	2	*(postCondition)*
	.	.	.	i	i	.	.	.	.	.	.	.	.	.	.	.	.	.		e an element of S
	.	.	.	i	i	.	i	.	.	.	.	.	.	.	.	.	.	.		e an element of E-S

Fig. 5.2.3.6 Data flow for Anneal algorithm

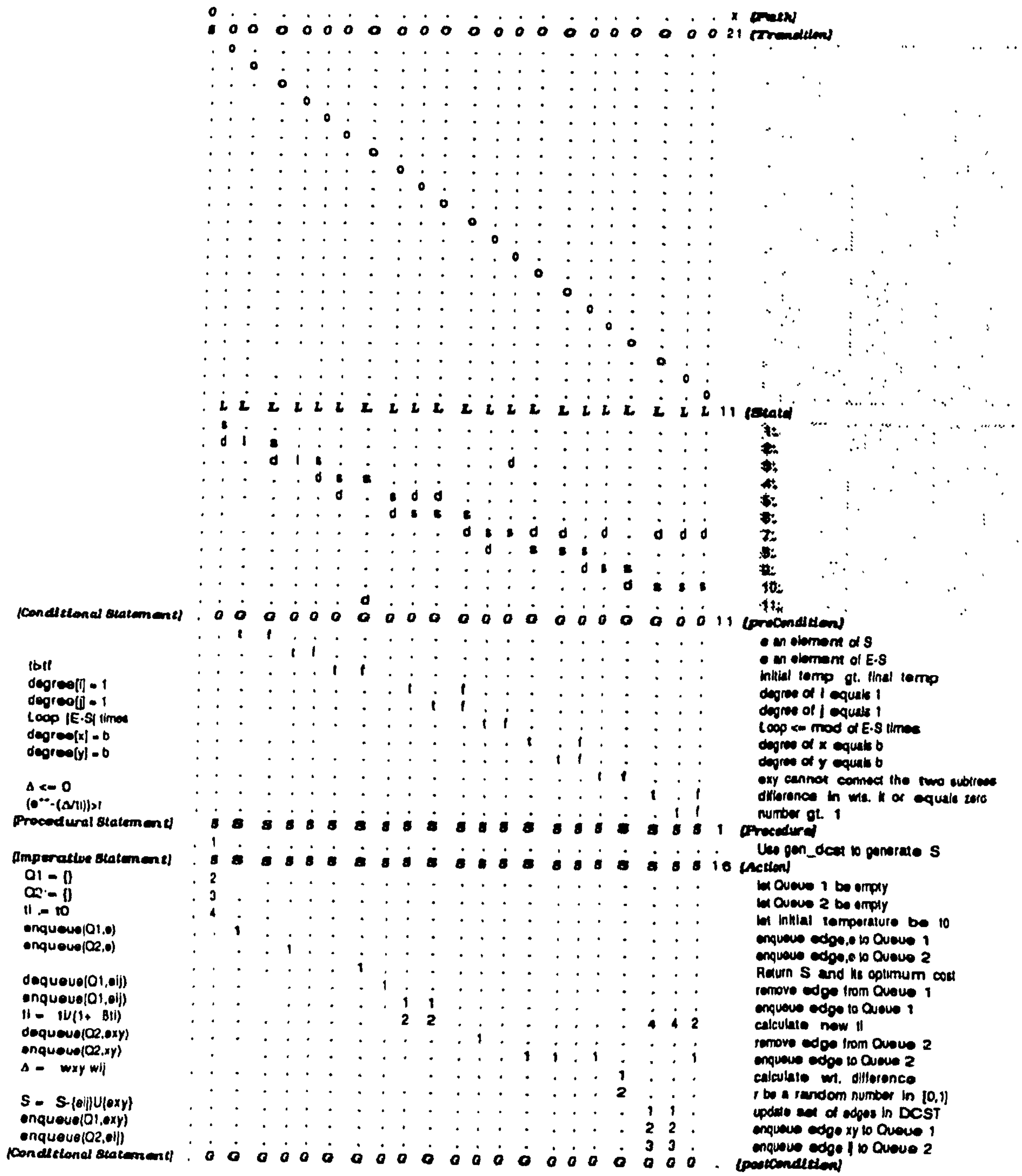
a) source code: original (see page 158)

1: Use gen_dcst() to generate S for a DCST.
Initialize queues Q_1 and Q_2 to be empty. Let $t_i = t_0$.
2: For all $e \in S$, enqueue(Q_1, e).
3: for all $e \in E - S$, enqueue(Q_2, e).
4: While $t_i > t_f$ do
begin
5: dequeue(Q_1, e_{ij}).
6: If degree[i] = 1 or degree[j] = 1 then
enqueue(Q_1, e_{ij}), let $t_i = t_i/(1 + \beta t_i)$, goto 5.
7: Loop $|E - S|$ times
begin
dequeue(Q_2, e_{xy}).
8: If degree[x] = b or degree[y] = b then enqueue(Q_2, e_{xy}), goto 7.
9: If e_{xy} cannot connect the two subtrees created by removing
e_{ij} from S, then enqueue(Q_2, e_{xy}), goto 7.
Let $\Delta = w_{xy} - w_{ij}$, r be a random number in $[0, 1]$.
10: If $\Delta \leq 0$ or $e^{-\Delta/t_i} > r$ then
begin
Let $S = S - \{e_{ij}\} \cup \{e_{xy}\}$.
enqueue(Q_1, e_{xy}), enqueue(Q_2, e_{ij}).
Let $t_i = t_i/(1 + \beta t_i)$.
end
enqueue(Q_2, e_{xy}), let $t_i = t_i/(1 + \beta t_i)$.
end {loop}
end
11: Return S and its cost for the best solution encountered.

b) edited source code: line nos removed, states 1..11 identified

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21		
	o																						x	{Path}
	s	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	21	{Transition}
		o																						
			o																					
				o																				
					o																			
						o																		
							o																	
								o																
									o															
										o														
											o													
												o												
													o											
														o										
															o									
																o								
																	o							
																		o						
																			o					
																				o				
																					o			
																						o		
		L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	11	{State}
		s																						1:
		d	l	s																				2:
				d	l	s								d										3:
						d	s	s																4:
							d		s	d	d													5:
									d	s	s	s												6:
												d	s	s	d	d		d		d	d	d		7:
													d		s	s	s							8:
																	d	s	s					9:
																			d	s	s	s		10:
								d																11:
{Conditional Statement}		C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	11	*{preCondition}*
			t	f																				e an element of S
					t	f																		e an element of E-S
ti>tf							t	f																initial temp gt. final temp
degree[i] = 1										t		f												degree of i equals 1
degree[j] = 1											t	f												degree of j equals 1
Loop |E-S| times													t	f										Loop <= mod of E-S times
degree[x] = b															t		f							degree of x equals b
degree[y] = b																t	f							degree of y equals b
																		t	f					exy cannot connect the two subtrees
Δ <= 0																				t		f		difference in wts. lt or equals zero
(e**-(Δ/ti))>r																					t	f		number gt 1
{Procedural Statement}		S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	1	*{Procedure}*
		1																						Use *gen_dcst* to generate S
{Imperative Statement}		S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	16	*{Action}*
Q1:= {}		2																						let Queue 1 be empty
Q2:= {}		3																						let Queue 2 be empty
ti := t0		4																						let initial temperature be t0
enqueue(Q1,e)			1																					enqueue edge,e to Queue 1
enqueue(Q2,e)					1																			enqueue edge,e to Queue 2
								1																Return S and its optimum cost
dequeue(Q1,eij)									1															remove edge from Queue 1
enqueue(Q1,eij)										1	1													enqueue edge to Queue 1
ti:= ti/(1+ βti)										2	2									4	4	2		calculate new ti
dequeue(Q2,exy)													1											remove edge from Queue 2
enqueue(Q2,xy)															1	1		1				1		enqueue edge to Queue 2
Δ:= wxy-wij																			1					calculate wt. difference
																			2					r be a random number in [0,1]
S:= S-{eij}U{exy}																				1	1			update set of edges in DCST
enqueue(Q1,exy)																				2	2			enqueue edge xy to Queue 1
enqueue(Q2,eij)																				3	3			enqueue edge ij to Queue 2
{Conditional Statement}		C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C		*{postCondition}*

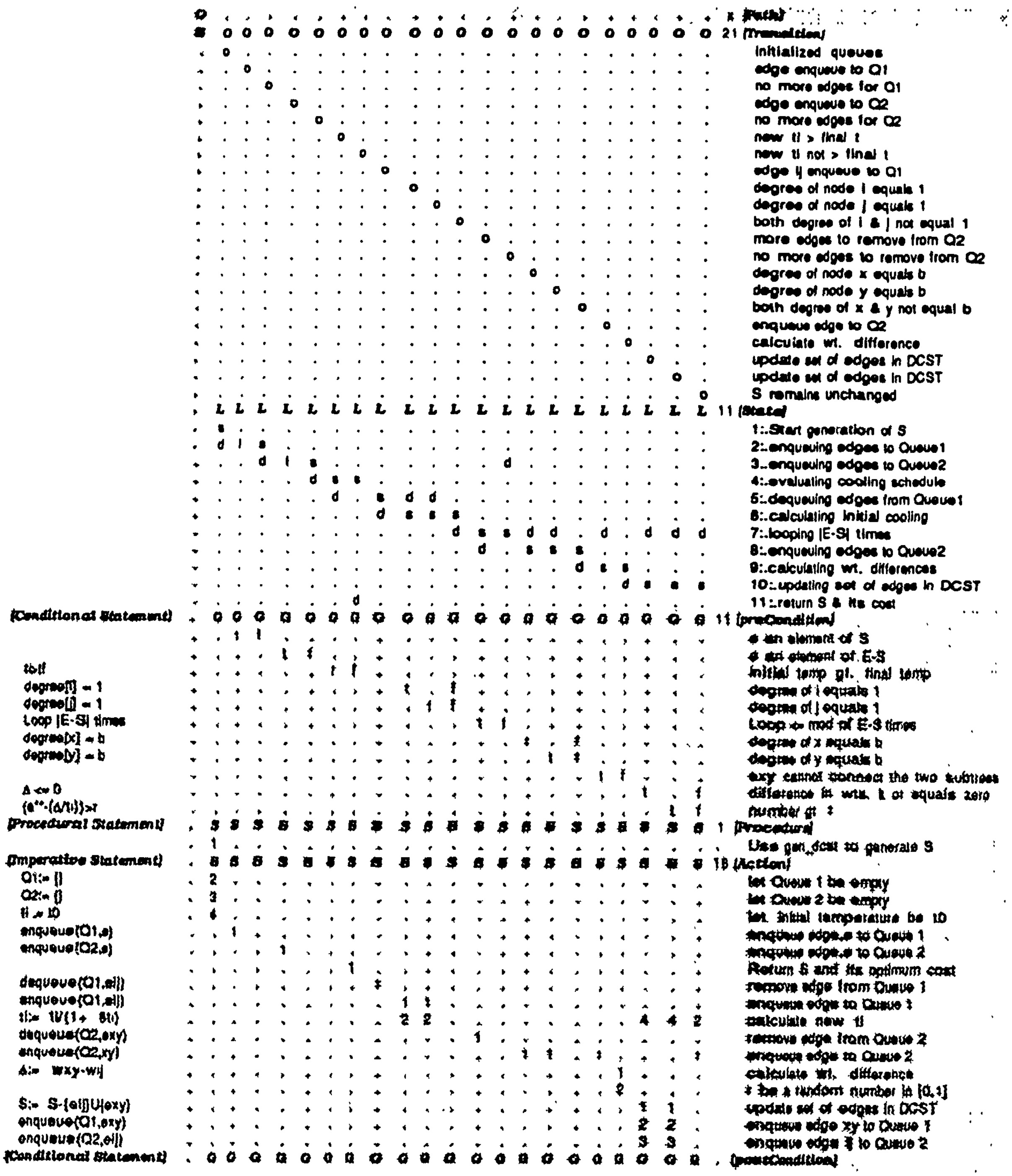
c) Normalized infoMap: shaded area to be filled in



c) Normalized infoMap: shaded area to be filled in

	0																						x	*(Path)*
	s	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	21	*(Transition)*
		o																						initialized queues
			o																					edge enqueue to Q1
				o																				no more edges for Q1
					o																			edge enqueue to Q2
						o																		no more edges for Q2
							o																	new ti > final t
								o																new ti not > final t
									o															edge ij enqueue to Q1
										o														degree of node i equals 1
											o													degree of node j equals 1
												o												both degree of i & j not equal 1
													o											more edges to remove from Q2
														o										no more edges to remove from Q2
															o									degree of node x equals b
																o								degree of node y equals b
																	o							both degree of x & y not equal b
																		o						enqueue edge to Q2
																			o					calculate wt. difference
																				o				update set of edges in DCST
																					o			update set of edges in DCST
																						o		S remains unchanged
		L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	11	*(State)*
		s																						1: Start generation of S
		d	i	s																				2: enqueuing edges to Queue1
				d	i	s								d										3: enqueuing edges to Queue2
						d	s	s																4: evaluating cooling schedule
							d		s	d	d													5: dequeuing edges from Queue1
									d	s	s	s												6: calculating initial cooling
												d	s	s	d	d		d		d	d	d		7: looping |E-S| times
													d		s	s	s							8: enqueuing edges to Queue2
																	d	s	s					9: calculating wt. differences
																			d	s	s	s		10: updating set of edges in DCST
								d																11: return S & its cost
(Conditional Statement)		G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	11	*(preCondition)*
			t	f																				e an element of S
					t	f																		e an element of E-S
ti>tf							t	f																initial temp gt. final temp
degree[i] = 1										t		f												degree of i equals 1
degree[j] = 1											t	f												degree of j equals 1
Loop |E-S| times													t	f										Loop <= mod of E-S times
degree[x] = b															t		f							degree of x equals b
degree[y] = b																t	f							degree of y equals b
																		t	f					exy cannot connect the two subtrees
$\Delta <= 0$																				t		f		difference in wts lt or equals zero
$(e^{-(\Delta/ti)})>r$																					t	f		number gt 1
(Procedural Statement)		S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	1	*(Procedure)*
		1																						Use gen_dcst to generate S
(Imperative Statement)		S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	16	*(Action)*
Q1 := {}		2																						let Queue 1 be empty
Q2 := {}		3																						let Queue 2 be empty
ti := t0		4																						let initial temperature be t0
enqueue(Q1,e)			1																					enqueue edge,e to Queue 1
enqueue(Q2,e)					1																			enqueue edge,e to Queue 2
								1																Return S and its optimum cost
dequeue(Q1,eij)									1															remove edge from Queue 1
enqueue(Q1,eij)										1	1													enqueue edge to Queue 1
ti := ti/(1+ βti)										2	2									4	4	2		calculate new ti
dequeue(Q2,exy)													1											remove edge from Queue 2
enqueue(Q2,xy)															1	1		1				1		enqueue edge to Queue 2
Δ := wxy-wij																			1					calculate wt. difference
																			2					r be a random number in [0,1)
S := S-{eij}U{exy}																				1	1			update set of edges in DCST
enqueue(Q1,exy)																				2	2			enqueue edge xy to Queue 1
enqueue(Q2,eij)																				3	3			enqueue edge ij to Queue 2
(Conditional Statement)		G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G		*(postCondition)*

d) Normalized InfoMap: states and transitions added (shaded area)



e) Normalized InfoMap: state diagram (unshaded area)

	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x		*(Path)*
	s	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	21		*(Transition)*
	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.			Initialized queues
	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.			edge enqueue to Q1
	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.			no more edges for Q1
	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.			edge enqueue to Q2
	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.			no more edges for Q2
	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.			new ti > final t
	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.			new ti not > final t
	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.			edge ij enqueue to Q1
	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.			degree of node i equals 1
	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.			degree of node j equals 1
	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.			both degree of i & j not equal 1
	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.			more edges to remove from Q2
	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.			no more edges to remove from Q2
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.			degree of node x equals b
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.			degree of node y equals b
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.			both degree of x & y not equal b
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.			enqueue edge to Q2
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.			calculate wt. difference
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.			update set of edges in DCST
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o			update set of edges in DCST
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o		S remains unchanged
	.	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	11	*(State)*
	.	s	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.			1:.Start generation of S
	.	d	i	s	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.			2:.enqueuing edges to Queue1
	.	.	.	d	i	s	.	.	.	.	.	.	.	d	.	.	.	.	.	.	.			3:.enqueuing edges to Queue2
	.	.	.	.	.	d	s	s	.	.	.	.	.	.	.	.	.	.	.	.	.			4..evaluating cooling schedule
	.	.	.	.	.	.	d	.	s	d	d	.	.	.	.	.	.	.	.	.	.			5:.dequeuing edges from Queue1
	.	.	.	.	.	.	.	.	d	s	s	s	.	.	.	.	.	.	.	.	.			6:.calculating initial cooling
	.	.	.	.	.	.	.	.	.	.	.	d	s	s	d	d	.	d	.	d	d	d		7..looping |E S| times
	.	.	.	.	.	.	.	.	.	.	.	.	d	.	s	s	s	.	.	.	.			8: enqueuing edges to Queue2
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	d	s	s	.	.			9 calculating wt. differences
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	d	s	s	s		10. updating set of edges in DCST
	.	.	.	.	.	.	.	d	.	.	.	.	.	.	.	.	.	.	.	.	.			11:.return S & its cost
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	11	*(preCondition)*
	.	.	t	f	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.			e an element of S
	.	.	.	.	t	f	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.			e an element of E-S
tb>tf	.	.	.	.	.	.	t	f	.	.	.	.	.	.	.	.	.	.	.	.	.			initial temp gt. final temp
degree[i] = 1	.	.	.	.	.	.	.	.	.	t	.	f	.	.	.	.	.	.	.	.	.			degree of i equals 1
degree[j] = 1	.	.	.	.	.	.	.	.	.	.	t	f	.	.	.	.	.	.	.	.	.			degree of j equals 1
Loop |E-S| times	.	.	.	.	.	.	.	.	.	.	.	.	t	f	.	.	.	.	.	.	.			Loop <= mod of E-S times
degree[x] = b	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	f	.	.	.	.			degree of x equals b
degree[y] = b	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	f	.	.	.	.			degree of y equals b
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	f	.	.	.		exy cannot connect the two subtrees
$\Delta <= 0$	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	.	f		difference in wts lt or equals zero
$(e^{**}\cdot(\Delta/ti))>r$	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	t	f		number gt 1
(Procedural Statement)	.	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	1	*(Procedural)*
	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		Use gen_dcst to generate S
(Imperative Statement)	.	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	16	*(Action)*
Q1:= {}	.	2	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		let Queue 1 be empty
Q2:= {}	.	3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		let Queue 2 be empty
ti := t0	.	4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		let initial temperature be t0
enqueue(Q1,e)	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		enqueue edge,e to Queue 1
enqueue(Q2,e)	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.		enqueue edge,e to Queue 2
	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.		Return S and its optimum cost
dequeue(Q1,eij)	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.	.		remove edge from Queue 1
enqueue(Q1,eij)	.	.	.	.	.	.	.	.	.	1	1	.	.	.	.	.	.	.	.	.	.	.		enqueue edge to Queue 1
ti:= ti/(1+ Bti)	.	.	.	.	.	.	.	.	.	2	2	.	.	.	.	.	.	.	.	4	4	2		calculate new ti
dequeue(Q2,exy)	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.		remove edge from Queue 2
enqueue(Q2,xy)	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	1	.	1	.	.	.	1		enqueue edge to Queue 2
Δ:= wxy-wij	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.	.		calculate wt difference
	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2	.	.	.		r be a random number in [0,1]
S:= S-{eij}U{exy}	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	1	.		update set of edges in DCST
enqueue(Q1,exy)	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	2	2	.		enqueue edge xy to Queue 1
enqueue(Q2,eij)	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	3	3	.		enqueue edge ij to Queue 2
(Conditional Statement)	.	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G		*(postCondition)*

f) Normalized infoMap to be optimized: shaded areas

```
                          O . . . . . . . . . . . . . . . .  x  (Path)
                          ■ O O O O O O O O O O O O O O O O  16 (Transition)
                          . o . . . . . . . . . . . . . . .     initialized queues
                          . . o . . . . . . . . . . . . . .     edge enqueue to Q1
                          . . . o . . . . . . . . . . . . .     edge enqueue to Q2
                          . . . . o . . . . . . . . . . . .     new ti not > final t
                          . . . . . o . . . . . . . . . . .     new ti > final t
                          . . . . . . o . . . . . . . . . .     edge ij enqueue to Q1
                          . . . . . . . o . . . . . . . . .     both degree of i & j not equal 1
                          . . . . . . . . o . . . . . . . .     either degree of i or j equals 1
                          . . . . . . . . . o . . . . . . .     more edges to remove from Q2
                          . . . . . . . . . . o . . . . . .     no more edges to remove from Q2
                          . . . . . . . . . . . o . . . . .     both degree of x & y not equal b
                          . . . . . . . . . . . . o . . . .     either degree of x or y equals b
                          . . . . . . . . . . . . . o . . .     enqueue edge to Q2
                          . . . . . . . . . . . . . . o . .     calculate wt. difference
                          . . . . . . . . . . . . . . . o .     S remains unchanged
                          . . . . . . . . . . . . . . . . o     update set of edges in DCST
                          . L L L L L L L L L L L L L L L L  11 (State)
                          . s . . . . . . . . . . . . . . .     1:.Start generation of S
                          . d l . . . . . . . . . . . . . .     2:.enqueuing edges to Queue1
                          . . e l . . . . . . d . . . . . .     3:.enqueuing edges to Queue2
                          . . . e s s . . . . . . . . . . .     4:.evaluating cooling schedule
                          . . . . . d s . d . . . . . . . .     5:.dequeuing edges from Queue1
                          . . . . . . d s s . . . . . . . .     6:.calculating initial cooling
                          . . . . . . . d . s s . d d . d d     7:.looping |E-S| times
                          . . . . . . . . . d . s s . . . .     8:.enqueuing edges to Queue2
                          . . . . . . . . . . . d . s s . .     9:.calculating wt. differences
                          . . . . . . . . . . . . . . d s s     10:.updating set of edges in DCST
                          . . . . d . . . . . . . . . . . .     11:.return S & its cost
(Conditional Statement)   . G G G G G G G G G G G G G G G G  9  (preCondition)
  tb>tl                   . . . . f t . . . . . . . . . . .     initial temp gt. final temp
  degree[i] = 1           . . . . . . . f c . . . . . . . .     degree of i equals 1
  degree[j] = 1           . . . . . . . f c . . . . . . . .     degree of j equals 1
  Loop |E-S| times        . . . . . . . . . f f . . . . . .     Loop <= mod of E-S times
  degree[x] = b           . . . . . . . . . . . f c . . . .     degree of x equals b
  degree[y] = b           . . . . . . . . . . . f c . . . .     degreeof y equals b
                          . . . . . . . . . . . . . f f . .     exy cannot connect the two subtrees
  Δ <= 0                  . . . . . . . . . . . . . . . f c     difference in wts. lt or equals zero
  (e**-(Δ/ti))>r          . . . . . . . . . . . . . . . f c     number gt. 1
(Procedural Statement)    . S S S S S S S S S S S S S S S S  1  (Procedure)
                          . 1 . . . . . . . . . . . . . . .     Use gen_dcst to generate S
(Imperative Statement)    . S S S S S S S S S S S S S S S S  16 (Action)
  Q1:= {}                 . 2 . . . . . . . . . . . . . . .     let Queue 1 be empty
  Q2.= {}                 . 3 . . . . . . . . . . . . . . .     let Queue 2 be empty
  ti .= t0                . 4 . . . . . . . . . . . . . . .     let initial temperature be t0
  enqueue(Q1,e)           . . 1 . . . . . . . . . . . . . .     enqueue edge,e to Queue 1
  enqueue(Q2,e)           . . . 1 . . . . . . . . . . . . .     enqueue edge,e to Queue 2
                          . . . . 1 . . . . . . . . . . . .     Return S and its optimum cost
  dequeue(Q1,eij)         . . . . . . 1 . . . . . . . . . .     remove edge from Queue 1
  enqueue(Q1,eij)         . . . . . . . . 1 . . . . . . . .     enqueue edge to Queue 1
  ti:= ti/(1+ βti)        . . . . . . . . 2 . . . . . . 2 4     calculate new ti
  dequeue(Q2,exy)         . . . . . . . . . 1 . . . . . . .     remove edge from Queue 2
  enqueue(Q2,xy)          . . . . . . . . . . . . 1 1 . 1 .     enqueue edge to Queue 2
  Δ.= wxy-wij             . . . . . . . . . . . . . . 1 . .     calculate wt. difference
                          . . . . . . . . . . . . . . 2 . .     r be a random number in [0,1]
  S = S-{eij}U{exy}       . . . . . . . . . . . . . . . . 1     update set of edges in DCST
  enqueue(Q1,exy)         . . . . . . . . . . . . . . . . 2     enqueue edge xy to Queue 1
  enqueue(Q2,eij)         . . . . . . . . . . . . . . . . 3     enqueue edge ij to Queue 2
(Conditional Statement)   . G G G G G G G G G G G G G G G G  2  (postCondition)
                          . . 1 . . . . . . . . . . . . . .     e an element of S
                          . . . t . . . . . . . . . . . . .     e an element of E-S
```

g) Optimized and documented InfoMap

Note: Steps - Optimized state diagram as Structured English (generated) and source code (generated) - are to be done as in Fig. 5.2.4.1

Fig. 5.2.4.5 Control flow for Anneal algorithm

Appendix V

Skeletonization of Binary patterns algorithm

This appendix contains the transformation of one procedure and two functions, namely, SKELETONIZE(), EDGEPOINT() and SAFEPOINT() respectively.

```
procedure SKELETONIZE(var PATTERN : pat_type;
                j, first row, last_row : integer; var d : integer);
var
     row :      integer;
     column :   integer;
     p :        pointer;
                {the variable p is used when referring to the point
                PATTTERN[row, column].}
     n :        8-neighbours;
                {the variables n[0] to n[7] are used when referring to the
                point PATTERN[row, column].}
     n :        8-neighbours   {the variables n[0] to n[7] are used when
                               referring to the 8-neighbours of the point p.}
     border · border_type;
            {Indicates which 4-neighbour caused the point p to become an edgepoint.}
begin
     for row := first_row to last_row do
        for column := 1 to MAXCOLUMN do
            begin
            if DARK(p) then
            {a point is considered to be DARK if it has the
            value ZERO. ie. it is not a safepoint.}

                  if EDGEPPOINT(n[j],n[j+4],border) then
                  {test each dark point to see if it is an edgepoint.}

                  begin
                  if SAFEPOINT(n,border) then
                  {Test each edgepoint to see whether it is
                  a safepoint. If it is a safepoint, then the point
                  is labelled by the value i. Otherwise, the point
                  becomes a flagged point and is labelled by the value
                  (i - MAXINT).}
                         p := i          {a safepoint}
                  else
                         p := i  - MAXINT; {a flagged point}
                  ADJUST(p,row,column);
                  {The procedure ADJUST, will be used only by the
                  data decomposition implementation. However it
                  has been included here so that only one version
                  of the procedure SKELETONIZE needs to be presented.}
                  end
                  else
                  d := d + 1;
                  {The point is a dark point which is neither flagged
                  nor declared to be a safepoint, so we increase our
                  counter d.}
            end;
end;
```

a) source code: original

```
procedure SKELETONIZE(var PATTERN : pat_type;
                j, first row, last_row : integer; var d : integer);
var
    row :       integer;
    column :    integer;
    p :         pointer;
                {the variable p is used when referring to the point
                PATTTERN[row, column].}
    n :         8-neighbours;
                {the variables n[0] to n[7] are used when referring to the
                point PATTERN[row, column].}
    n :         8-neighbours    {the variables n[0] to n[7] are used when
                                referring to the 8-neighbours of the point p.}
    border : border_type;
            {Indicates which 4-neighbour caused the point p to become an edgepoint.}
begin
1:      for row := first_row to last_row do
2:          for column := 1 to MAXCOLUMN do
                begin
3:              if DARK(p) then
                {a point is considered to be DARK if it has the
                value ZERO. ie. it is not a safepoint.}

4:                      if EDGEPPOINT(n[j],n[j+4],border) then
                        {test each dark point to see if it is an edgepoint }

                        begin
5:                      if SAFEPOINT(n,border) then
                        {Test each edgepoint to see whether it is
                        a safepoint. If it is a safepoint, then the point
                        is labelled by the value i. Otherwise, the point
                        becomes a flagged point and is labelled by the value
                        (i - MAXINT).}
                                p := i          {a safepoint}
                        else
                                p := i - MAXINT; {a flagged point}
                        ADJUST(p,row,column);
                        {The procedure ADJUST, will be used only by the
                        data decomposition implementation. However it
                        has been included here so that only one version
                        of the procedure SKELETONIZE needs to be presented }
                        end
                        else
                        d := d + 1;
                        {The point is a dark point which is neither flagged
                        nor declared to be a safepoint, so we increase our
                        counter d.}
                end;
    end;
6:
```

b) edited source code: states 1 .. 6 identified

	O	.	.	.	.	.	.	.	.	.	.	x	*{Path}*
	S	O	O	O	O	O	O	O	O	O	O	10	*{Transition}*
	.	o	.	.	.	.	.	.	.	.	.		
	.	.	o	.	.	.	.	.	.	.	.		
	.	.	.	o	.	.	.	.	.	.	.		
	.	.	.	.	o	.	.	.	.	.	.		
	.	.	.	.	.	o	.	.	.	.	.		
	.	.	.	.	.	.	o	.	.	.	.		
	.	.	.	.	.	.	.	o	.	.	.		
	.	.	.	.	.	.	.	.	o	.	.		
	.	.	.	.	.	.	.	.	.	o	.		
	.	.	.	.	.	.	.	.	.	.	o		
	.	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	6	*{State}*
	.	s	s	d	.	.	.	.	.	.	.		1:
	.	.	d	s	s	d	.	d	.	d	d		2:
	.	.	.	.	d	s	s	.	.	.	.		3:
	.	.	.	.	.	.	d	s	s	.	.		4:
	.	.	.	.	.	.	.	.	d	s	s		5:
	.	d	.	.	.	.	.	.	.	.	.		6:
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	5	*{preCondition}*
row := first_row to last_row	.	f	t	.	.	.	.	.	.	.	.		within permissible rows
column := 1 to MAXCOLUMN	.	.	.	f	t	.	.	.	.	.	.		within permissible columns
DARK(p)	.	.	.	.	.	f	t	.	.	.	.		point is DARK
EDGEPPOINT(n[j],n[j+4],border)	.	.	.	.	.	.	.	f	t	.	.		dark point is an edgepoint
SAFEPOINT(n,border)	.	.	.	.	.	.	.	.	.	f	t		edgepoint is a safepoint
{Imperative Statement}	.	S	S	S	S	S	S	S	S	S	S	3	*{Action}*
d := d + 1;	.	.	.	.	.	.	.	1	.	.	.		increase counter d
p := i - MAXINT	.	.	.	.	.	.	.	.	.	1	.		label point by value of i-MAXINT
p := i	.	.	.	.	.	.	.	.	.	.	1		label point by value of i
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	.	*{postCondition}*

c) Normalized infoMap: shaded area to be filled in

	O	.	.	.	.	.	.	.	.	.	.	x	*{Path}*
	S	O	O	O	O	O	O	O	O	O	O	10	*{Transition}*
	.	o	.	.	.	.	.	.	.	.	.		no more rows
	.	.	o	.	.	.	.	.	.	.	.		try next row
	.	.	.	o	.	.	.	.	.	.	.		no more columns
	.	.	.	.	o	.	.	.	.	.	.		try next column
	.	.	.	.	.	o	.	.	.	.	.		point is not dark
	.	.	.	.	.	.	o	.	.	.	.		point is dark
	.	.	.	.	.	.	.	o	.	.	.		dark point is not an edgepoint
	.	.	.	.	.	.	.	.	o	.	.		dark point is an edgepoint
	.	.	.	.	.	.	.	.	.	o	.		edgepoint is not a safepoint
	.	.	.	.	.	.	.	.	.	.	o		edgepoint is a safepoint
	.	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*L*	6	*{State}*
	.	s	s	d	.	.	.	.	.	.	.		1: start: row testing
	.	.	d	s	s	d	.	d	.	d	d		2: column testing
	.	.	.	.	d	s	s	.	.	.	.		3: testing for dark point
	.	.	.	.	.	.	d	s	s	.	.		4: testing for edgepoint
	.	.	.	.	.	.	.	.	d	s	s		5: testing for safepoint
	.	d	.	.	.	.	.	.	.	.	.		6: exit
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	5	*{preCondition}*
row := first_row to last_row	.	f	t	.	.	.	.	.	.	.	.		within permissible rows
column := 1 to MAXCOLUMN	.	.	.	f	t	.	.	.	.	.	.		within permissible columns
DARK(p)	.	.	.	.	.	f	t	.	.	.	.		point is DARK
EDGEPPOINT(n[j],n[j+4],border)	.	.	.	.	.	.	.	f	t	.	.		dark point is an edgepoint
SAFEPOINT(n,border)	.	.	.	.	.	.	.	.	.	f	t		edgepoint is a safepoint
{Imperative Statement}	.	S	S	S	S	S	S	S	S	S	S	3	*{Action}*
d := d + 1,	.	.	.	.	.	.	.	1	.	.	.		increase counter d
p := i - MAXINT	.	.	.	.	.	.	.	.	.	1	.		label point by value of i-MAXINT
p := i	.	.	.	.	.	.	.	.	.	.	1		label point by value of i
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	.	*{postCondition}*

d) Normalized infoMap: states and transitions added (shaded area)

	O	.	.	.	.	.	.	.	.	.	.	x	*{Path}*
	S	O	O	O	O	O	O	O	O	O	O	10	*{Transition}*
	.	o	.	.	.	.	.	.	.	.	.		no more rows
	.	.	o	.	.	.	.	.	.	.	.		try next row
	.	.	.	o	.	.	.	.	.	.	.		no more columns
	.	.	.	.	o	.	.	.	.	.	.		try next column
	.	.	.	.	.	o	.	.	.	.	.		point is not dark
	.	.	.	.	.	.	o	.	.	.	.		point is dark
	.	.	.	.	.	.	.	o	.	.	.		dark point is not an edgepoint
	.	.	.	.	.	.	.	.	o	.	.		dark point is an edgepoint
	.	.	.	.	.	.	.	.	.	o	.		edgepoint is not a safepoint
	.	.	.	.	.	.	.	.	.	.	o		edgepoint is a safepoint
	.	L	L	L	L	L	L	L	L	L	L	6	*{State}*
	.	s	s	d	.	.	.	.	.	.	.		1:.start row testing
	.	.	d	s	s	d	.	d	.	d	d		2..column testing
	.	.	.	.	d	s	s	.	.	.	.		3:.testing for dark point
	.	.	.	.	.	.	d	s	s	.	.		4:.testing for edgepoint
	.	.	.	.	.	.	.	.	d	s	s		5:.testing for safepoint
	.	d	.	.	.	.	.	.	.	.	.		6: exit
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	5	*{preCondition}*
row := first_row to last_row	.	f	t	.	.	.	.	.	.	.	.		within permissible rows
column := 1 to MAXCOLUMN	.	.	.	f	t	.	.	.	.	.	.		within permissible columns
DARK(p)	.	.	.	.	.	f	t	.	.	.	.		point is DARK
EDGEPPOINT(n[j],n[j+4],border)	.	.	.	.	.	.	.	f	t	.	.		dark point is an edgepoint
SAFEPOINT(n,border)	.	.	.	.	.	.	.	.	.	f	t		edgepoint is a safepoint
{Imperative Statement}	.	S	S	S	S	S	S	S	S	S	S	3	*{Action}*
d := d + 1;	.	.	.	.	.	.	.	1	.	.	.		increase counter d
p := i - MAXINT	.	.	.	.	.	.	.	.	.	1	.		label point by value of i-MAXINT
p := i	.	.	.	.	.	.	.	.	.	.	1		label point by value of i
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	.	*{postCondition}*

e) Normalized InfoMap: state diagram (unshaded area)

	O	.	.	.	.	.	.	.	.	.	.	x	*{Path}*
	S	O	O	O	O	O	O	O	O	O	O	10	*{Transition}*
	.	o	.	.	.	.	.	.	.	.	.		no more rows
	.	.	o	.	.	.	.	.	.	.	.		try next row
	.	.	.	o	.	.	.	.	.	.	.		no more columns
	.	.	.	.	o	.	.	.	.	.	.		try next column
	.	.	.	.	.	o	.	.	.	.	.		point is not dark
	.	.	.	.	.	.	o	.	.	.	.		point is dark
	.	.	.	.	.	.	.	o	.	.	.		dark point is not an edgepoint
	.	.	.	.	.	.	.	.	o	.	.		dark point is an edgepoint
	.	.	.	.	.	.	.	.	.	o	.		edgepoint is not a safepoint
	.	.	.	.	.	.	.	.	.	.	o		edgepoint is a safepoint
	.	L	L	L	L	L	L	L	L	L	L	6	*{State}*
	.	s	s	d	.	.	.	.	.	.	.		1:.start: row testing
	.	.	d	s	s	d	.	d	.	d	d		2: column testing
	.	.	.	.	d	s	s	.	.	.	.		3:.testing for dark point
	.	.	.	.	.	.	d	s	s	.	.		4:.testing for edgepoint
	.	.	.	.	.	.	.	.	d	s	s		5: testing for safepoint
	.	d	.	.	.	.	.	.	.	.	.		6:.exit
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	5	*{preCondition}*
row := first_row to last_row	.	f	t	.	.	.	.	.	.	.	.		within permissible rows
column := 1 to MAXCOLUMN	.	.	.	f	t	.	.	.	.	.	.		within permissible columns
DARK(p)	.	.	.	.	.	f	t	.	.	.	.		point is DARK
EDGEPPOINT(n[j],n[j+4],border)	.	.	.	.	.	.	.	f	t	.	.		dark point is an edgepoint
SAFEPOINT(n,border)	.	.	.	.	.	.	.	.	.	f	t		edgepoint is a safepoint
{Imperative Statement}	.	S	S	S	S	S	S	S	S	S	S	3	*{Action}*
d := d + 1,	.	.	.	.	.	.	.	1	.	.	.		increase counter d
p := i - MAXINT	.	.	.	.	.	.	.	.	.	1	.		label point by value of i-MAXINT
p := i	.	.	.	.	.	.	.	.	.	.	1		label point by value of i
{Conditional Statement}	.	G	G	G	G	G	G	G	G	G	G	.	*{postCondition}*

f) Normalized InfoMap to be optimized: shaded area

	O	.	.	.	.	.	.	.	x	*(Path)*
	S	O	O	O	O	O	O	O	7	*(Transition)*
	.	o	.	.	.	.	.	.		no more rows
	.	.	o	.	.	.	.	.		permissible row
	.	.	.	o	.	.	.	.		found safepoint
	.	.	.	.	o	.	.	.		found edgepoint
	.	.	.	.	.	o	.	.		point is dark
	.	.	.	.	.	.	o	.		try next column
	.	.	.	.	.	.	.	o		try next row
	.	L	L	L	L	L	L	L	3	*(State)*
	.	s	s	.	.	.	.	d		1:.start: row testing
	.	.	d	l	l	l	l	s		2:.processing of pattern
	.	d	.	.	.	.	.	.		3:.exit
(Conditional Statement)	.	G	G	G	G	G	G	G	5	*(preCondition)*
row := first_row to last_row	.	f	t	.	.	.	.	.		within permissible rows
column := 1 to MAXCOLUMN	.	.	.	t	t	t	t	f		within permissible columns
DARK(p)	.	.	.	t	t	t	f	.		point is DARK
EDGEPPOINT(n[j],n[j+4],border)	.	.	.	t	t	f	.	.		dark point is an edgepoint
SAFEPOINT(n,border)	.	.	.	t	f	.	.	.		edgepoint is a safepoint
(Imperative Statement)	.	S	S	S	S	S	S	S	3	*(Action)*
p := i	.	.	.	1	.	.	.	.		label point by value of i
p := i - MAXINT;	.	.	.	.	1	.	.	.		label point by value of i-MAXINT
d := d + 1;	.	.	.	.	.	1	.	.		increase counter d
(Conditional Statement)	.	G	G	G	G	G	G	G	.	*(postCondition)*

g) Optimized and documented infoMap

Note: Steps - Optimized state diagram as Structured English and source code (generated) - are to be done as in Fig. 5.3.4.1

Fig. 5.3.4.2 Control flow for procedure SKELETONIZE

```
function EDGEPOINT(n[j],n[j+4] : pointer;
        var border : border_type) : boolean;

        {A point p, is considered to be WHITE if it satisfies the
        following condition :
                (value of p) < (i - MAXINT).
        That is, the point is an original white point, or
        it is a flagged point.

        The variable border, returns the value indicating which
        boolean expression S[border] should be tested, (where
        border = 0, 2, 4, 6), to detect safepoints.}

begin
        if WHITE(n[j]) then
                {Test for either a right or top edgepoint.}
        begin
                border := j;
                EDGEPOINT := TRUE;
        end
        else if WHITE(n[j+4]) then
                {Test for either a left or bottom edgepoint.}
        begin
                border := j + 4;
                EDGEPOINT := TRUE;
        end
        else
                EDGEPOINT := FALSE;
  end;
```

a) source code: original

```
        function EDGEPOINT(n[j],n[j+4] : pointer;
              var border : border_type) : boolean;

              {A point p, is considered to be WHITE if it satisfies the
              following condition :
                    (value of p) < (i - MAXINT).
              That is, the point is an original white point, or
              it is a flagged point.

              The variable border, returns the value indicating which
              boolean expression S[border] should be tested,  (where
              border = 0, 2, 4, 6), to detect safepoints.}

        begin
1:            if WHITE(n[j]) then
                    {Test for either a right or top edgepoint.}
              begin
                    border := j;
                    EDGEPOINT := TRUE;
              end
2:            else if WHITE(n[j+4]) then
                    {Test for either a left or bottom edgepoint.}
              begin
                    border := j + 4;
                    EDGEPOINT := TRUE;
              end
              else
                    EDGEPOINT := FALSE;
         end;
3:
```

b) edited source code: states 1 .. 3 identified

	O	.	.	.	.	x	*{Path}*
	S	*O*	*O*	*O*	*O*	4	*{Transition}*
	.	o	.	.	.		
	.	.	o	.	.		
	.	.	.	o	.		
	.	.	.	.	o		
	.	*L*	*L*	*L*	*L*	3	*{State}*
		s	s	.	.		1:.
		.	d	s	s		2:.
	.	d	.	d	d		3:.
{Conditional Statement}	.	*G*	*G*	*G*	*G*	2	*{preCondition}*
WHITE(n[j])	.	t	f	.	.		either a right or top edgepoint
WHITE(n[j+4])	.	.	.	t	f		either a left or bottom edgepoint
{Imperative Statement}	.	**S**	**S**	**S**	**S**	4	*{Action}*
border := j;	.	1	.	.	.		let border be right or top edgepoint
border := j+4;	.	.	.	1	.		let border be left or bottom edgepoint
EDGEPOINT := TRUE;	.	2	.	2	.		set edgepoint to true
EDGEPOINT := FALSE;	.	.	.	.	1		set edgepoint to false
{Conditional Statement}	.	*G*	*G*	*G*	*G*	.	*{postCondition}*

c) Normalized infoMap: shaded area to be filled in

	O	.	.	.	.	x	*{Path}*
	S	*O*	*O*	*O*	*O*	4	*{Transition}*
		o	.				right or top edgepoint
	.	.	o	.	.		not a right or top edgepoint
	.	.		o	.		left or bottom edgepoint
			.		o		not an edgepoint
	.	*L*	*L*	*L*	*L*	3	*{State}*
	.	s	s	.	.		1:.start: testing for right or top edgepoint
		.	d	s	s		2:.testing for left or bottom edgepoint
		d	.	d	d		3:.exit
{Conditional Statement}	.	*G*	*G*	*G*	*G*	2	*{preCondition}*
WHITE(n[j])	.	t	f	.	.		either a right or top edgepoint.
WHITE(n[j+4])	.	.	.	t	f		either a left or bottom edgepoint.
{Imperative Statement}	.	**S**	**S**	**S**	**S**	4	*{Action}*
border := j;	.	1	.	.	.		let border be right or top edgepoint
border := j+4;		.	.	1	.		let border be left or bottom edgepoint
EDGEPOINT := TRUE;	.	2	.	2	.		set edgepoint to true
EDGEPOINT := FALSE;	.	.		.	1		set edgepoint to false
{Conditional Statement}	.	*G*	*G*	*G*	*G*		*{postCondition}*

d) Normalized infoMap: states and transition added (shaded area)

	O	.	.	.	.	x	***{Path}***
	S	O	O	O	O	4	***{Transition}***
	.	o		.	.		right or top edgepoint
	.	.	o	.	.		not a right or top edgepoint
	.	.	.	o	.		left or bottom edgepoint
	.		.	.	o		not an edgepoint
	.	L	L	L	L	3	***{State}***
	.	s	s	.	.		1:.start: testing for right or top edgepoint
	.	.	d	s	s		2:.testing for left or bottom edgepoint
	.	d	.	d	d		3:.exit
{Conditional Statement}	.	G	G	G	G	2	***{preCondition}***
WHITE(n[j])	.	t	f	.	.		either a right or top edgepoint.
WHITE(n[j+4])	.	.	.	t	f		either a left or bottom edgepoint.
{Imperative Statement}	.	S	S	S	S	4	***{Action}***
border := j;	.	1	.	.	.		let border be right or top edgepoint
border := j+4;	.	.	.	1	.		let border be left or bottom edgepoint
EDGEPOINT := TRUE;	.	2	.	2	.		set edgepoint to true
EDGEPOINT := FALSE;	.	.	.	.	1		set edgepoint to false
{Conditional Statement}	.	G	G	G	G	.	***{postCondition}***

e) Normalized infoMap: state diagram (unshaded area)

	O	.	.	.	.	x	***{Path}***
	S	O	O	O	O	4	***{Transition}***
		o	.	.	.		right or top edgepoint
	.	.	o	.	.		not a right or top edgepoint
	.	.	.	o	.		left or bottom edgepoint
		.	.	.	o		not an edgepoint
	.	L	L	L	L	3	***{State}***
		s	s	.	.		1:.start: testing for right or top edgepoint
	.	.	d	s	s		2:.testing for left or bottom edgepoint
	.	d	.	d	d		3:.exit
{Conditional Statement}	.	G	G	G	G	2	***{preCondition}***
WHITE(n[j])	.	t	f	.	.		either a right or top edgepoint.
WHITE(n[j+4])		.	.	t	f		either a left or bottom edgepoint.
{Imperative Statement}	.	S	S	S	S	4	***{Action}***
border := j,	.	1	.	.	.		let border be right or top edgepoint
border := j+4;		.	.	1	.		let border be left or bottom edgepoint
EDGEPOINT := TRUE;		2	.	2	.		set edgepoint to true
EDGEPOINT := FALSE;	.	.	.	.	1		set edgepoint to false
{Conditional Statement}	.	G	G	G	G	.	***{postCondition}***

f) Normalized infoMap to be optimized: (shaded area)

	O	.	.	.	x	*{Path}*
	S	*O*	*O*	*O*	3	*{Transition}*
	.	o	.	.		right or top edgepoint
	.	.	o	.		left or bottom edgepoint
	.	.	.	o		not an edgepoint
	.	*L*	*L*	*L*	2	*{State}*
	.	s	s	s		1:.start processing point
	.	d	d	d		2:.exit processing point
{Conditional Statement}	.	*G*	*G*	*G*	2	*{preCondition}*
WHITE(n[j])	.	t	f	f		either a right or top edgepoint.
WHITE(n[j+4])	.	.	t	f		either a left or bottom edgepoint.
{Imperative Statement}	.	*S*	*S*	*S*	4	*{Action}*
border := j;	.	1	.	.		let border be right or top edgepoint
border := j+4;	.	.	1	.		let border be left or bottom edgepoint
EDGEPOINT := TRUE;	.	2	2	.		set edgepoint to true
EDGEPOINT := FALSE;	.	.	.	1		set edgepoint to false
{Conditional Statement}	.	*G*	*G*	*G*	.	*{postCondition}*

g) Optimized and documented infoMap

Note: Steps - Optimized state diagram as Structured English (generated) and source code (generated) - are to be done as in Fig. 5.3.4.1

Fig. 5.3.4.3 Control flow for function EDGEPOINT

```
function SAFEPOINT(n : 8-neighbours; border : border_type) : boolean;
    {This function evaluates the appropriate safepoint boolean expression and
    returns TRUE if the point is a safepoint and FALSE if it is not.}
      begin
            case border of
                  0 : {Evaluate for a right safepoint.}
                      SAFEPOINT := not(n[4]  (n[5] + n[6] + n[2] + n[3])
                                   (n[6] + not(n[7]))  (n[2] + not(n[1]));
                  2 : {Evaluate for a top safepoint.}
                      SAFEPOINT := not(n[6]  (n[7] + n[0] + n[4] + n[5])
                                   (n[0] + not(n[1]))  (n[4] + not(n[3]));
                  4 : {Evaluate for a left safepoint.}
                      SAFEPOINT := not(n[0]  (n[1] + n[2] + n[6] + n[7])
                                   (n[2] + not(n[3]))  (n[6] + not(n[5]));
                  6 : {Evaluate for a bottom safepoint.}
                      SAFEPOINT := not(n[2]  (n[3] + n[4] + n[0] + n[1])
                                   (n[4] + not(n[5]))  (n[0] + not(n[7]));
            end;    {case}
      end
```

a) source code: original

```
    function SAFEPOINT(n : 8-neighbours; border : border_type) : boolean;
        {This function evaluates the appropriate safepoint boolean expression and
        returns TRUE if the point is a safepoint and FALSE if it is not.}
          begin
                case border of
1:                    0 : {Evaluate for a right safepoint.}
                          SAFEPOINT := not(n[4]  (n[5] + n[6] + n[2] + n[3])
                                       (n[6] + not(n[7]))  (n[2] + not(n[1]));
2:                    2 : {Evaluate for a top safepoint.}
                          SAFEPOINT := not(n[6]  (n[7] + n[0] + n[4] + n[5])
                                       (n[0] + not(n[1]))  (n[4] + not(n[3]));
3:                    4 : {Evaluate for a left safepoint.}
                          SAFEPOINT := not(n[0]  (n[1] + n[2] + n[6] + n[7])
                                       (n[2] + not(n[3]))  (n[6] + not(n[5]));
4:                    6 : {Evaluate for a bottom safepoint.}
                          SAFEPOINT := not(n[2]  (n[3] + n[4] + n[0] + n[1])
                                       (n[4] + not(n[5]))  (n[0] + not(n[7]));
                end;    {case}
          end
5:
```

b) edited source code: states 1 .. 5 identified

```
                                                      O  .  .  .  .  .  .  .  .  x  {Path}
                                                      S  O  O  O  O  O  O  O  O  8  {Transition}
                                                         o
                                                            o
                                                               o
                                                                  o
                                                                     o
                                                                        o
                                                                           o
                                                                              o
                                                      .  L  L  L  L  L  L  L  L  5  {State}
                                                         s  s                          1:.
                                                            d  s  s                    2
                                                                  d  s  s              3'.
                                                                        d  s  s        4'.
                                                         d     d     d     d  d        5'.
{Conditional Statement}                               .  G  G  G  G  G  G  G  G  4  {preCondition}
  border = 0                                             t  t                          right safepoint
  border = 2                                                   t  f                    top safepoint
  border = 4                                                         t  t              left safepoint
  border = 6                                                               t  t        bottom safepoint
{Imperative Statement}                                .  S  S  S  S  S  S  S  S  4  {Action}
  SAFEPOINT = not(n[4]  (n[5] + n[6] + n[2] + n[3])      1                             evaluate right safepoint
                 (n[6] + not(n[7]))  (n[2] + not(n[1])).
  SAFEPOINT = not(n[6]  (n[7] + n[0] + n[4] + n[5])            1                       evaluate top safepoint
                 (n[0] + not(n[1]))  (n[4] + not(n[3])).
  SAFEPOINT = not(n[0]  (n[1] + n[2] + n[6] + n[7])                  1                 evaluate left safepoint
                 (n[2] + not(n[3]))  (n[6] + not(n[5])).
  SAFEPOINT = not(n[2]  (n[3] + n[4] + n[0] + n[1])                        1           evaluate bottom safepoint
                 (n[4] + not(n[5]))  (n[0] + not(n[7]))
{Conditional Statement}                               G  G  G  G  G  G  G  G  G     {postCondition}
```

c) Normalized infoMap: shaded area to be filled in

```
                                                      O  .  .  .  .  .  .  .  .  x  {Path}
                                                      S  O  O  O  O  O  O  O  O  8  {Transition}
                                                         o                             right safepoint
                                                            o                          not a right safepoint
                                                               o                       top safepopint
                                                                  o                    not a top safepoint
                                                                     o                 left safepoint
                                                                        o              not a left safopoint
                                                                           o           bottom safepoint
                                                                              o        not a bottom safepoint
                                                      .  L  L  L  L  L  L  L  L  5  {State}
                                                         s  s                          1'.start evaluation of right safepoint
                                                            d  s  s                    2' evaluation of top safepoint
                                                                  d  s  s              3' evaluation of left safepoint
                                                                        d  s  s        4 evaluation of bottom safepoint
                                                         d     d     d     d  d        5:.exit
{Conditional Statement}                               .  G  G  G  G  G  G  G  G  4  {preCondition}
  border = 0                                             t  t                          right safepoint
  border = 2                                                   t  t                    top safepoint
  border = 4                                                         t  t              left safepoint
  border = 6                                                               t  t        bottom safepoint
{Imperative Statement}                                .  S  S  S  S  S  S  S  S  4  {Action}
  SAFEPOINT = not(n[4]  (n[5] + n[6] + n[2] + n[3])      1                             evaluate right safepoint
                 (n[6] + not(n[7]))  (n[2] + not(n[1])).
  SAFEPOINT = not(n[6]  (n[7] + n[0] + n[4] + n[5])            1                       evaluate top safepoint
                 (n[0] + not(n[1]))  (n[4] + not(n[3])).
  SAFEPOINT = not(n[0]  (n[1] + n[2] + n[6] + n[7])                  1                 evaluate left safepoint
                 (n[2] + not(n[3]))  (n[6] + not(n[5])).
  SAFEPOINT = not(n[2]  (n[3] + n[4] + n[0] + n[1])                        1           evaluate bottom safepoint
                 (n[4] + not(n[5]))  (n[0] + not(n[7])).
{Conditional Statement}                               G  G  G  G  G  G  G  G  G     {postCondition}
```

d) Normalized infoMap: states and transitions added (shaded area)

	O	.	.	.	.	.	.	.	.	x	(Path)
	S	O	O	O	O	O	O	O	O	8	(Transition)
	.	o									right safepoint
	.		o								not a right safepoint
	.			o							top safepopint
	.			.	o						not a top safepoint
	.					o					left safepoint
	.						o				not a left safepoint
	.							o			bottom safepoint
	.					.			o		not a bottom safepoint
	.	L	L	L	L	L	L	L	L	5	(State)
	.	s	s								1 start evaluation of right safepoint
	.		d	s	s						2 evaluation of top safepoint
	.				d	s	s		.		3 evaluation of left safepoint
	.	.					d	s	s		4 evaluation of bottom safepoint
	.	d	.	d		d		d	d		5 exit
(Conditional Statement)	.	G	G	G	G	G	G	G	G	4	(preCondition)
border = 0		t	f	.	.	.	.	.	.		right safepoint
border = 2		.	.	t	f	.	.	.			top safepoint
border = 4		.	.	.	.	t	f				left safepoint
border = 6		.	.	.	.	.	.	t	f		bottom safepoint
(Imperative Statement)	.	S	S	S	S	S	S	S	S	4	(Action)
SAFEPOINT = not(n[4] . (n[5] + n[6] + n[2] + n[3]) (n[6] + not(n[7])) (n[2] + not(n[1])),	.	1	.	.	.	.	.	.	.	.	evaluate right safepoint
SAFEPOINT = not(n[6] . (n[7] + n[0] + n[4] + n[5]) (n[0] + not(n[1])) . (n[4] + not(n[3])),		.	.	1	.	.	.	.			evaluate top safepoint
SAFEPOINT = not(n[0] . (n[1] + n[2] + n[6] + n[7]) (n[2] + not(n[3])) (n[6] + not(n[5]));		.	.	.	.	1	.	.			evaluate left safepoint
SAFEPOINT = not(n[2] . (n[3] + n[4] + n[0] + n[1]) (n[4] + not(n[5])) . (n[0] + not(n[7]));		.	.	.	.	.	.	.	1	.	evaluate bottom safepoint
(Conditional Statement)	G	G	G	G	G	G	G	G	G	.	(postCondition)

e) Normalized infoMap: state diagram (unshaded area)

	O	.	.	.	.	.	.	.	.	x	(Path)
	S	O	O	O	O	O	O	O	O	8	(Transition)
		o			.	.	.	.	.		right safepoint
	.	.	o	.		.	.	.	.		not a right safepoint
		.	.	o			.	.	.		top safepopint
		.	.	.	o	.	.	.	.		not a top safepoint
		.	.	.	.	o	.	.	.		left safepoint
	.	.	.	.	.	.	o	.	.		not a left safepoint
		.	.	.	.	.	.	o	.		bottom safepoint
		.	.	.	.	.	.	.	o		not a bottom safepoint
	.	L	L	L	L	L	L	L	L	5	(State)
		s	s	.	.	.	.	.	.		1 start evaluation of right safepoint
		.	d	s	s	.	.	.	.		2 evaluation of top safepoint
		.			d	s	s	.	.		3 evaluation of left safepoint
						.	d	s	s		4 evaluation of bottom safepoint
		d		d		d	.	d	d		5 exit
(Conditional Statement)	.	G	G	G	G	G	G	G	G	4	(preCondition)
border = 0		t	f	.	.	.	.	.	.		right safepoint
border = 2		.	.	t	f	.	.				top safepoint
border = 4		.	.	.	.	t	f				left safepoint
border = 6		.	.	.	.	.	.	t	f		bottom safepoint
(Imperative Statement)	.	S	S	S	S	S	S	S	S	4	(Action)
SAFEPOINT = not(n[4] (n[5] + n[6] + n[2] + n[3]) (n[6] + not(n[7])) (n[2] + not(n[1])).		1	.	.	.	.	.	.	.		evaluate right safepoint
SAFEPOINT = not(n[6] (n[7] + n[0] + n[4] + n[5]) (n[0] + not(n[1])) (n[4] + not(n[3])).		.	.	1	.	.	.	.	.		evaluate top safepoint
SAFEPOINT = not(n[0] (n[1] + n[2] + n[6] + n[7]) (n[2] + not(n[3])) (n[6] + not(n[5])).		.	.	.	.	1	.	.	.		evaluate left safepoint
SAFEPOINT = not(n[2] (n[3] + n[4] + n[0] + n[1]) (n[4] + not(n[5])) (n[0] + not(n[7])).		.	.	.	.	.	.	1	.		evaluate bottom safepoint
(Conditional Statement)	G	G	G	G	G	G	G	G	G		(postCondition)

f) Normalized infoMap to be optimized: shaded area

	O	.	.	.	.	.	x	*{Path}*
	S	O	O	O	O	O	5	*{Transition}*
	.	o	.	.	.	.		try right safepoint
	.	.	o	.	.	.		try top safepopint
	.	.	.	o	.	.		try left safepoint
	.	.	.	.	o	.		try bottom safepoint
	.	.	.	.	.	o		incorrect data
	.	L	L	L	L	L	2	*{State}*
	.	s	s	s	s	s		1:.start processing safepoint
	.	d	d	d	d	d		2:.exit processing safepoint
{Conditional Statement}	.	G	G	G	G	G	4	*{preCondition}*
border = 0	.	t	.	.	.	f		right safepoint
border = 2	.	.	t	.	.	f		top safepoint
border = 4	.	.	.	t	.	f		left safepoint
border = 6	.	.	.	.	t	f		bottom safepoint
{Imperative Statement}	.	S	S	S	S	S	4	*{Action}*
SAFEPOINT := not(n[4] . (n[5] + n[6] + n[2] + n[3]) . (n[6] + not(n[7])) . (n[2] + not(n[1]));	.	1	.	.	.	.		evaluate right safepoint
SAFEPOINT := not(n[6] . (n[7] + n[0] + n[4] + n[5]) . (n[0] + not(n[1])) . (n[4] + not(n[3]));	.	.	1	.	.	.		evaluate top safepoint
SAFEPOINT := not(n[0] . (n[1] + n[2] + n[6] + n[7]) . (n[2] + not(n[3])) . (n[6] + not(n[5]));	.	.	.	1	.	.		evaluate left safepoint
SAFEPOINT := not(n[2] . (n[3] + n[4] + n[0] + n[1]) . (n[4] + not(n[5])) . (n[0] + not(n[7]));	.	.	.	.	1	.		evaluate bottom safepoint
{Conditional Statement}	G	G	G	G	G	G	.	*{postCondition}*

g) Optimized and documented infoMap

Note: Steps - Optimized state diagram as Structured English (generated) and source code (generated) - are to be done as in Fig. 5.3.4.1

Fig. 5.3.4.4 Control flow for function SAFEPOINT