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.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

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.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.
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

© Thompson Cummings, 1991
The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

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

## Chapter 1  Introduction 1
1.1 Purpose 1
1.2 State of Art 2
1.3 TAPi methodology 2
1.4 Hierarchical model 2
1.5 Data model 3
1.6 Data flow 3
1.7 Control flow 3
1.8 Inheritance tracing 3
1.9 Reusability 4
1.10 Expressiveness 4
1.11 Organization of the Thesis 4

## Chapter 2  infoMaps 6
2.1 Introduction 6
2.2 infoSchemas 6
2.3 An infoMap example 7

## Chapter 3  The TAPi methodology 9
3.1 Introduction 9
3.2 Structuring select views 10
3.3 Hierarchical view 12
3.4 Data model view 13
3.5 Data flow view 14
3.6 Control flow view 14

## Chapter 4  Program Normalization and Optimization 17
4.1 Introduction 17
4.2 Models of Control Flow Structures 19
4.3 Programming Style 26
4.4 Constructs, Styles and Documentation 28
4.5 Conclusions 30

## Chapter 5  Application of TAPi 32
5.1 String Search algorithms 32
5.1.1 Introduction 32
5.1.2 Hierarchy, Data model and data flow 34
5.1.3 Control flow 36
5.1.4 Summary and Conclusions 44
5.2 Degree-Constrained Minimum Spanning Tree algorithms 48
5.2.1 Introduction 48
5.2.2 Data model 49
5.2.3 Data flow 50
5.2.4 Control flow 52
5.2.5 Summary 62
5.3 Skeletonization of Binary Patterns algorithm 63
  5.3.1 Introduction 63
  5.3.2 Data model and data flow 64
  5.3.3 Recreating ONE_PROCESSOR_SPTA using comments only 66
  5.3.4 Control flow 69

Chapter 6 Required Properties of the TAPI Environment 76
  6.1 Productivity of Knowledge Acquisition 76
  6.2 Quality of TAPI products 76
    6.2.1 Verification 76
    6.2.2 Validation 82
  6.3 Interface 82
    6.3.1 Graphic User Interface 83
    6.3.2 Dynamic infoMap (Animation) 83

Chapter 7 Conclusions 85

References 87

Appendices 91
  Appendix I Syntax of infoMaps notation 91
  Appendix II Program Normalization and Optimization 92
  Appendix III String Search Algorithms 94
  Appendix IV Degree-Constrained Minimum Spanning Tree algorithms 137
  Appendix V Skeletonization of Binary Patterns algorithm 166
## ABBREVIATIONS

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

Fig. 1.11.1  Reading Strategy  5

Fig. 2.2.1  Generic infoSchema for a sequential program  6

Fig. 2.3.1  Binary Search program  7

Fig. 3.2.1  Selected views used by TAPi  10

Fig. 3.2.2  A knowledge acquisition process  11

Fig. 3.2.3  A Straight Forward search algorithm  12

Fig. 3.3.1  Hierarchical view used by TAPi  12

Fig. 3.4.1  Data model view used by TAPi  13

Fig. 3.4.2  Data model of Straight Forward search algorithm  13

Fig. 3.5.1  Data flow view used by TAPi  14

Fig. 3.5.2  Data flow of a Straight Forward search algorithm  14

Fig. 3.6.1  Control flow view used by TAPi  15

Fig. 4.1.2  Generic control flow graphs  18

Fig. 4.2.1  Programming constructs  20

Fig. 4.3.1  All zero row program  27

Fig. 4.4.1  Program attributes for all-zero row program  27

Fig. 5.1.1.1  String Search Problem Statement (Narrative)  33

Fig. 5.1.2.1  Hierarchical view of string search algorithms  34

Fig. 5.1.2.2  Data model for string search algorithms  35
List of Figures (cont.)

Fig. 5.1.2.3  Data flow within a Straight Forward search algorithm  36
Fig. 5.1.3.1  An infoSchema of the control flow of string search algorithms  37
Fig. 5.1.3.2  Control flow for a Straight Forward search algorithm  38
Fig. 5.1.3.11 A Control flows (code column hidden) for string search optimized algorithms  43
Fig. 5.1.4.1  Data flows for string search algorithms  44
Fig. 5.1.4.2  Attributes of string search algorithms  45
Fig. 5.1.4.3  Common attributes of string search algorithms  46
Fig. 5.2.1.1  DCMST Problem Statement (Narrative)  48
Fig. 5.2.2.1  Data model for DCMST algorithms  50
Fig. 5.2.3.1  gen_dcest algorithm  51
Fig. 5.2.3.2  infoSchema of the data flow for the gen_dcest algorithm:
   bold area  51
Fig. 5.2.3.3  Data flow for gen_dcest algorithm  52
Fig. 5.2.4.1  Control flow for gen_dcest algorithm  53
Fig. 5.3.1.1  Skeletonization of Binary patterns Problem
   Statement (Narrative)  63
Fig. 5.3.1.2  infoSchema for the Skeletonization of binary patterns  64
Fig. 5.3.2.1  Data model for the skeletonization of binary patterns
   algorithm  65
Fig. 5.3.2.2  Data flow for the skeletonization of binary patterns algorithm  65
List of Figures (cont.)

Fig. 5.3.3.1 Recreating ONE_PROCESSOR_SPTA using comments only 66

Fig. 5.3.4.1 Control flow for ONE_PROCESSOR_SPTA procedure 69

Fig. 6.2.1.1 Verification of the Straight Forward search algorithm 78

Fig. 6.2.1.2 Path verification for SF algorithm: pattern "in"
in text "infoMap" 80

Fig. 6.2.1.3 Path verification for SF algorithm: pattern "Map" in text "infoMap" 81

Figures of Appendix I

Syntax of infoMaps notation 91

Figures of Appendix II

Fig. 4.3.2 All-zero row Program 92

Fig. 4.3.3 All-zero row Program in EPN style 93

Figures of Appendix III

Fig. 5.1.12 Control flow for computing table for KMP pattern matching 95

Fig. 5.1.13 Control flow of Knuth-Morris-Pratt string search algorithm 100

Fig. 5.1.14 Control flow for computing DELTA1 and DELTA2 105

Fig. 5.1.15 Control flow of Boyer-Moore string search algorithm 112

Fig. 5.1.16 Control flow for computing table TD1[] 117
Fig. 5.1.17  Control flow for computing table TD2[]  121
Fig. 5.1.18  Control flow for a Substring Search algorithm  125
Fig. 5.1.19  Control flow for a Quick Search algorithm  132

Figures of Appendix IV

Fig. 5.2.3.4  Data flow for Primal algorithm  138
Fig. 5.2.4.2  Control flow for Primal algorithm  139
Fig. 5.2.3.5  Data flow for Dual algorithm  146
Fig. 5.2.4.3  Control flow for Dual algorithm (main)  147
Fig. 5.2.4.4  Control flow for Dual algorithm (procedure)  153
Fig. 5.2.3.6  Data flow for Anneal algorithm  159
Fig. 5.2.4.5  Control flow for Anneal algorithm  160

Figures of Appendix V

Fig. 5.3.4.2  Control flow procedure SKELETONIZE  167
Fig. 5.3.4.3  Control flow for function EDGEPOINT  172
Fig. 5.3.4.4  Control flow for function SAFEPOINT  177
Chapter 1

Introduction

1.1 Purpose:

This thesis is aimed at providing a methodology for enabling Software Engineers to 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 Transformation of Algorithms and Programs with infoMaps(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.

![Graph representation of reading sequence]

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.

\[
\begin{align*}
O & \quad \text{(Transition)} \\
L & \quad \text{(State)} \\
G & \quad \text{(preCondition)} \\
S & \quad \text{(Action)} \\
G & \quad \text{(postCondition)}
\end{align*}
\]

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: each transition links states; each transition is implemented by a sequence of actions guarded by preconditions and asserted 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-true, f-false and c-complementary.

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:  else if 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;
```

```
a) Pascal-type code  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 source and "d" destination. 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.

\[
\begin{array}{cccccccc}
V & . & . & . & . & . & . & 1: \text{Hierarchy} \\
. & V & V & . & . & . & . & : \text{Data Model} \\
. & . & V & V & . & . & . & : \text{Data Flow} \\
. & . & . & . & V & V & . & : \text{Control Flow} \\
H & O & O & . & . & . & x & \text{Algorithm} \\
\{\text{Declarative Statement}\} & O & . & . & . & x & \text{Type} \\
\{\text{Declarative Statement}\} & M & M & F & O & . & x & \text{Data-object/Attribute} \\
. & . & . & . & O & . & x & \text{Path} \\
. & . & . & . & F & S & O & \text{Transition} \\
. & . & . & . & F & L & x & \text{State} \\
\{\text{Conditional Statement}\} & F & G & . & . & . & . & \text{preCondition} \\
\{\text{Procedural Statement}\} & H & . & F & . & S & x & \text{Procedure} \\
\{\text{Imperative Statement}\} & . & F & . & S & x & \text{Action} \\
\{\text{Conditional Statement}\} & . & F & G & . & . & x & \text{postCondition} \\
\end{array}
\]

©W.M. Jaworski 1990

Fig. 3.2.1 Selected views used by TAPi

The third column shows a O-ne to M-many relationship between algorithm and data-objects. There is also, shown by fourth column, a O-ne to M-many 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.

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. \( \text{return("pattern found at", i)} \)
8. \( \text{else do} \)
9. \( j := j+1; \)
10. \( k := k+1; \)
11. \( \text{end; do} \)
12. \( \text{return("pattern not found");} \)

**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.

![Diagram of hierarchical view](image)

**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.

![Diagram]

**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.

![Diagram]

**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.

\[
\begin{array}{c|c|c}
\text{A} & \text{A} & 4 \text{ [View]} \\
v & v & 3: \text{Data Flow} \\
o & x & \text{[Algorithm]} \\
(D) & o & \text{[Data-object/Attribute]} \\
. & f & \text{[Transition]} \\
. & f & \text{[State]} \\
(C) & f & \text{[preCondition]} \\
(P) & f & \text{[Procedure]} \\
(I) & f & \text{[Action]} \\
(C) & f & \text{[postCondition]} \\
\end{array}
\]

**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.

\[
\begin{array}{c|c|c}
o & 1 & \text{[Algorithm]} \\
0 & \text{Straight Forward} & \text{[Data-object/Attribute]} \\
(D) & 7 & \text{[Data-object/Attribute]} \\
i & b & \text{guess ptr. for pat within the text} \\
n & l & \text{length of text string} \\
m & l & \text{length of pattern substring} \\
PAT[] & l & \text{pattern substring} \\
TXT[] & l & \text{text string} \\
"pattern found at" & o & \text{success message} \\
"pattern not found" & o & \text{failure message} \\
\end{array}
\]

**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.
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 source, destination, loop, assertion and exception respectively.

By examining control flow components of the infoMaps in Fig. 4.2.1, it is noticeable that all control flow graphs are built 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             Pancode notation             EPN notation
if V then begin
    S1;
    S2
end;

Control flow graphs

infoMap notation

Pascal-type notation             Pancode notation             EPN notation
if V1 then
    S1
else if V2 then begin
    S2
    S3
    end
else
    S4;

Control flow graphs

infoMap notation

Fig. 4.2.1 Programming constructs
Pascal-type notation

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

Pancode notation

if V1
  S1
also if V2
  S2
also if V3
  S3

EPN notation

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

Control flow graphs

Conventional

Schematic

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

Pascal-type notation

While V do begin
  S1;
  S2
cend;

Pancode notation

if V
  S1
repeat

EPN notation

{V:S1;S2!EXIT}

Control flow graphs

Conventional

Schematic

(d) While...do...

Fig. 4.2.1 Programming constructs (cont.)
### Programming Constructs (cont.)

**Pascal-type notation**

While true do begin
  S1;
  S2
end;

**EPN notation**

\[
\{S1;S2\}
\]

**Control flow graphs**

Conventional

Schematic

---

**infoMap notation**

**Pascal-type notation**

for k:=first to last by step do begin
  S1
  S2
end;

**EPN notation**

\[
\{[\neg\text{EXIT}];S1;S2;\text{next } k\}
\]

where,

\[ V = k:=\text{first to last} \]

**Control flow graphs**

Conventional

Schematic

---

**infoMap notation**

**Pascal-type notation**

for k:=first to last do begin
  S1
  S2
end;

**EPN notation**

\[
\{[\neg\text{EXIT}]\}
\]

**Control flow graphs**

Conventional

Schematic

---

(c). Infinite loop

---

(F). For-loop

---

Fig. 4.2.1 Programming constructs (cont.)
Pascal-type notation | Pancode notation | EPN notation
---|---|---
repeat
  S1;
  S2
until not V | do
  S1
  S2
also if V
  repeat | \(\{S1;S2[-V:EXIT]\}\)

**Control flow graphs**

**infoMap notation**

Conventional | Schematic

---

Pascal-type notation | Pancode notation | EPN notation
---|---|---
S1;
while V do begin
  S2;
  S1
end | do
  S1
  S2
also if V
  repeat | \(\{S1[-V:EXIT];S2\}\)

**Control flow graphs**

**infoMap notation**

Conventional | Schematic

---

**Fig. 4.2.1 Programming constructs (cont.)**
Pascal-type notation

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

Pancode notation

if V1
   S1
also if V2
   S2
repeat

EPN notation

\([-V1:EXIT]S1[-V2:EXIT]S2\)

\(1\)

infoMap notation
"as is" infoMap

\(2\)

infoMap Notation
optimized infoMap

Control flow graphs

\((i)\) Loop, exit, exit

Pascal-type notation

end;

Pancode notation

Fig. 4.2.1 Programming constructs (cont.)
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 guaranteed 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 a more 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.

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

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 (e.g., 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. Trade-offs 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.

**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.
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".
### 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.
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 semantically 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. \(\quad i := i+1;\)
4. \(\quad j := i; k := 1;\)
5. \(\quad\text{while}(\text{PAT}(k) = \text{TXT}(j))\) do
6. \(\quad\quad\text{if}(k = m)\) then
7. \(\quad\quad\quad\text{return}("pattern found at", i)\)
8. \(\quad\quad\text{else}\) do
9. \(\quad\quad\quad j := j+1;\)
10. \(\quad\quad\quad k := k+1;\)
11. \(\quad\quad\text{end};\)
12. \(\text{end};\)
13. \(\text{return}("pattern not found");\)

\[\text{a) source code: original}\]

1 : \(i := 0;\)
2 : \(\text{while}(i<n-m+1)\) do
3 : \(\quad i := i+1;\)
4 : \(\quad j := i; k := 1;\)
5 : \(\quad\text{while}(\text{PAT}(k) = \text{TXT}(j))\) do
6 : \(\quad\quad\text{if}(k = m)\) then
7 : \(\quad\quad\quad\text{return}("pattern found at", i)\)
8 : \(\quad\quad\text{else}\) do
9 : \(\quad\quad\quad j := j+1;\)
10 : \(\quad\quad\quad k := k+1;\)
11 : \(\quad\quad\text{end};\)
12 : \(\text{end};\)
13 : \(\text{return}("pattern not found");\)

\[\text{b) edited source code: line nos removed, states 1..5 identified}\]
c) Normalized InfoMap: shaded areas to be filled in

---

---

d) Normalized InfoMap: states and transitions added (shaded areas)
\begin{enumerate}
\item Normalized InfoMap: state diagram (unshaded area)
\item Normalized InfoMap to be optimized: shaded area
\end{enumerate}
\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & x \\
6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\
\end{array}
\]

\textbf{[Path]} 
- Initialize guess ptr. for pat within the text
- location within the text
- location outside the text
- all pat chars matched
- match at current text location
- pat elem does not match with text elem

\textbf{[State]} 
1: Start
2: Evaluating remaining text
3: Matching pat elem with text elem
4: Exit

\textbf{[Conditional Statement]} 
\[ l = n-m+1 \]
\[ \text{PAT}(k) = \text{TXT}(l) \]
\[ k = m \]

\textbf{[Imperative Statement]} 
\[ i := 0; \]
\[ i := i+1 \]
\[ j := i; \]
\[ k := 1; \]
\[ \text{return}(\text{"pattern not found"}); \]
\[ \text{return}(\text{"pattern found at", l}) \]
\[ j := j+1; \]
\[ k := k+1; \]

\textbf{[Conditional Statement]} 
\[ 1 \]

\textbf{[preCondition]} 
- Location within text
- Pat elem equals text elem
- All pat chars compared

\textbf{[Action]} 
- Set guess ptr. to zero
- Increment guess ptr. by 1
- Set text ptr. to guess ptr. value
- Set pat ptr. to 1
- Pattern not found
- Pattern found at guess ptr. value
- Increment text ptr. by 1
- Increment pat ptr. by 1

\textit{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 optimized 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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Straight Forward</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quick Search</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search process</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>KMP Failure Table</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BM DELTA1 &amp; DELTA2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computing table TD1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computing table TD2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimized</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>branch</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>12</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>branch with fork</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>loop with exit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>transition</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>16</td>
<td>7</td>
<td>13</td>
<td>6</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>state</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>9</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>precondition</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>action</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>14</td>
<td>7</td>
<td>14</td>
<td>7</td>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>postCondition</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Straight Forward</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knuth-Morris-Pratt (KMP)</td>
<td>.</td>
<td>.</td>
<td>V</td>
<td>.</td>
<td>.</td>
<td>V</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boyer-Moore (BM)</td>
<td>.</td>
<td>.</td>
<td>V</td>
<td>.</td>
<td>.</td>
<td>V</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>V</td>
<td>V</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>Quick Search</td>
<td>.</td>
<td>.</td>
<td>V</td>
<td>.</td>
<td>.</td>
<td>V</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>V</td>
<td>V</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>Substring Search</td>
<td></td>
<td></td>
<td>V</td>
<td>V</td>
<td></td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimized</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Common</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>branch</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch with fork</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>loop with exit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>transition</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>state</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>preCondition</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>procedure</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>action</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>postCondition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 . . . . . . . . . . . . integer
real . . . . . . . . . . . . real number

V a a a . . . . . . the set of nodes
i a . a a a v . . 1, ..., n; nodes of V
init[] a a a . . . . . . Function; a node corresponding to j
j a 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 a v . . . . a node not in P
b a a a . a . . 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 a v . . summation of weights in a tree
n a a a a v . . no. of nodes in the graph
\infty . 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 a v . . . . a node of T
d[] a a a a . . degree of node
x . a a a v . . . a node of T_j
y . . a 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
I[] . . a . . . . . . final matrix
r . a . a . . . . . . a random number between 0 and 1
t . . . . a . . . . a temperature
I . . . . . . . . . . . . a reducer
\beta . . . a . a . . beta, a constant in temp calc.
delta . . . . . a a . . a difference between two weights
Q1 . . . . . . . . . . queue with edges already in S
Q2 . . . . . . . . . . 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. \( S \subseteq E. \)
\( \forall i \in V, \text{degree}[i] \) is the current degree of \( i \) in DCST.
\( \forall i \in V, \) assume \( w_{k_i} = \min_{e \in P} w_{e_i}, \) 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 \text{degree}[\text{ini}[k]] \geq b, \) then
\begin{align*}
&\forall j \in V - P, \) let \( w_{\text{ini}[j], j} = w_{j, \text{ini}[k]} = \infty. \\
&\forall j \in V - P, \) assume \( w_{g'j} = \min_{g \in P} w_{gj}, \) let \( u[j] = w_{g'j}, \) ini\([j]\) = \( g'. \)
\end{align*}
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_dest algorithm

The infoSchema of Fig. 3.5.1 is used in this section to transform the data
flow aspect of the gen_dest 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.

Fig. 5.2.3.2 infoSchema of the data flow for the gen_dest a algorithm:
bold area
Fig. 5.2.3.3 Data flow for gen_d cst 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_d cst 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_d cst 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

**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. \( S \subseteq E \).
\( \forall i \in V, \ \text{degree}[i] \) is the current degree of \( i \) in DCST.
\( \forall i \in V, \ \text{assume} \ w_{k'i} = \min_{k \in P} w_{ki}, \ \text{let} \ u[i] = w_{k'i}, \ \text{ini}[i] = k' \).

Algorithm: 1: Let \( P = \{1\}, \ S = \emptyset \).
2: \( \forall j \in V, \ \text{let} \ \text{ini}[j] = 1, \ u[j] = w_{1j} \).
3: Let \( u[k] = \min_{j \in V \setminus P} u[j] \).
4: If \( \text{degree}[\text{ini}[k]] \geq b \), then
   begin
   5: \( \forall j \in V \setminus P, \ \text{let} \ w_{\text{ini}[k], j} = w_{j, \text{ini}[k]} = \infty \).
   6: \( \forall j \in V \setminus P, \ \text{assume} \ w_{g'j} = \min_{g \in P} w_{gj}, \ \text{let} \ u[j] = w_{g'j}, \ \text{ini}[j] = g' \).
   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: \( \forall j \in V \setminus P, \ \text{if} \ w_{kj} < u[j], \ \text{then} \ \text{let} \ u[j] = w_{kj}, \ \text{ini}[j] = k \).
   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
d) Normalised infoMap: states and transitions added (shaded area)
e) Normalized infoknap: state diagram (unshaded area)
x (Path)

0
s 0 0 0 0 0 0 0 0 0 1
[Transition]
initialize weights
initialize nodes and edges
compute minimum wt.
degree of ln[i] => b
degree of ln[i] < b
wt. not considered
find min wt. of edge from P to V-P
weight of current edge < wt. of edge ||
weight of current edge >= wt. of edge ||
no more nodes of V-P left
all nodes in V examined

7
L L L L L L 7
l s
i d s e d s d l i s s d 1
e 1 1 1 1 1 1 1 1 1 1
[State]
1: Start initialization of weights
2: Finding k
3: Testing degree of node
4: Not considering weights anymore
5: Calculating new minimum weight
6: Comparing weights
7: Exit

5
G G G G G G G G G G G G G G G G G G 5
[Conditional Statement]
| an element of V
degree of ln[i] => b
P <= V
| an element of V-P
W[i] < s[i]

5
G G G G G G G G G G 5
[Conditional Statement]
degree of ln[i] => b
nodes in DCST not equal to V
| an element of V-P

15
S S S S S S S S S S S S S S S S 15
[Imperative Statement]
ln[i] := 1
U[i] := W[i]
P := 1
S := 0
U[k] := min U[i];
P := P U [k]
S := S U (ln[i],k,k,ln[i])
W[ln[k]] := w[i]
W[i], ln[k] := w[i]
W[g] := min(g elem. of P W[g])
U[i] := W[g]
ln[i] := g'
U[i] := W[i]
ln[i] := k

1
G G G G G G G G G G 1
[Conditional Statement]
| an element of V-P

1
G G G G G G G G G G 1
[postCondition]
| an element of V-P

(Algorithm)
Let initial node corresponding to | be 1
Let W[i] be the weight of edge | i
Set nodes already in DCST to 1
Set nodes already in DCST to 0
Set wt. of elem k to the min of U[i]
Update set of edges in DCST
Update set of edges in DCST
Let matrix wt. of edge not in P be infinity
Let transposed matrix wt. of that edge not in P be infinity
Let wt. of edge not in P be min of corresponding edge in P
Let U[i] be that wt. not in P
Let node corresponding to | be not in P
Let U[i] be wt. of current edge k
Let node corresponding to | be k
Return S and its cost

1
G G G G G G G G G G 1
[postCondition]
| an element of V-P

(Imperative Statement)
ln[i] := 1
U[i] := W[i]
P := 1
S := 0
U[k] := min U[i];
P := P U [k]
S := S U (ln[i],k,k,ln[i])
W[ln[k]] := w[i]
W[i], ln[k] := w[i]
W[g] := min(g elem. of P W[g])
U[i] := W[g]
ln[i] := g'
U[i] := W[i]
ln[i] := k

5
G G G G G G G G G G G G G G G G G G 5
[Conditional Statement]
degree of ln[i] => b
nodes in DCST not equal to V
| an element of V-P

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 inl[k] >= b, CONTINUE AT 4
   degree of inl[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 if, CONTINUE AT 6
   weight of current edge >= wt. of edge if, 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)
exit, CONTINUE AT 2
j not elem. of V-P -> goto 2

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

weight of current edge >= wt. of edge if, CONTINUE AT 6
P <> V and j elem. of V-P and WkJ >= U[i] -> 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

1) 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.

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

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 edgepoints
   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
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 safe points, 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;  \quad \text{[Initialize the scan number.]} \\
3: \quad \text{repeat} \\
    i := i + 1;  \quad \text{[Increment the pass number by one.]} \\
    j := 0;  \quad \text{[Set scan type to left/right endpoints.]} \\
    k := k + 1;  \quad \text{[Increment the scan number by one.]} \\
    \text{SKELETONIZE(PATTERN,} j, 1, \text{MAXROW,} d[k]);  \\
    \quad \text{(Execute the kth scan on the entire pattern.)} \\
4: \quad \text{if (} d[k] \neq 0 \text{) and (} d[k] \neq d[k-2] \text{) then} \\
    \quad \quad \text{(If criterion1 and criterion2 are FALSE,} \\
    \quad \quad \quad \text{then prepare for the next scan.)} \\
    \quad \text{begin} \\
    \quad \quad j := 2;  \quad \text{[Set scan type to top/bottom endpoints.]} \\
    \quad \quad k := k+1;  \quad \text{[Increment the scan number by one.]} \\
    \quad \text{SKELETONIZE(PATTERN,} j, 1, \text{MAXROW,} d[k]);  \\
    \quad \quad \text{(Execute the kth scan on the entire pattern.)} \\
    \quad \text{end;} \\
5: \quad \text{until (} d[k] = 0 \text{) or (} d[k] = d[k-2] \text{);}  \\
    \quad \quad \text{(Repeat executing passes on the entire pattern} \\
    \quad \quad \quad \text{until criterion1 or criterion2 is TRUE.)} \\
6: \quad \text{end;}

b) edited source code: states 1 .. 6 identified
(Conditional Statement)

k : = 1 to MAXSCAN
(d[k] <> 0)
(d[k] <> d[k-2])
(d[k] = 0)
(d[k] = d[k-2])

(Procedural Statement)

SKELETONIZE(PATTERN, j, MAXROW, d[k]);

(Action)

1 := 0;
1 := 1;
1 := 1;
2 := 0;
3 := k + 1;
2 := 1;
1 := 1;

(Conditional Statement)

G G G G G G G (postCondition)

(c) Normalized InfoMap: shaded areas to be filled in
73

\[
\begin{array}{c}
\text{(Path)} \\
\text{(Transition)} \\
\text{initialize pass no.} \\
\text{more dark points} \\
\text{no more dark points} \\
\text{try left/right scan} \\
\text{try top/bottom scan} \\
\text{suspend scan} \\
\text{continue scanning} \\
\text{stop scanning} \\
\text{(State)} \\
\text{1: Start} \\
\text{2: initialization of dark points} \\
\text{3: left/right scan} \\
\text{4: top/bottom scan} \\
\text{5: testing for end of scan} \\
\text{6: Exit} \\
\text{(preCondition)} \\
\text{scan permissible} \\
\text{dark pt.} \leq \text{zero} \\
\text{dark pt.} \geq \text{dark pt.} \leq \text{zero} \\
\text{dark pt.} \leq \text{zero} \\
\text{dark pt.} \geq \text{dark pt.} \leq \text{zero} \\
\text{(Procedure)} \\
\text{execute the kth scan on the entire pat.} \\
\text{(Action)} \\
\text{initialize the pass number} \\
\text{initialize d to ZERO} \\
\text{initialize the scan number} \\
\text{increment the pass number by one} \\
\text{set scan type to left/right edgpoints} \\
\text{increment the scan number by one} \\
\text{set scan type to top/bottom edgpoints} \\
\text{(postCondition)} \\
\end{array}
\]

\[
\begin{array}{c}
\text{(Conditional Statement)} \\
k := \text{-1 to MAXSCAN} \\
d[k] := 0 \\
d[k] := -k+2 \\
d[k] := 0 \\
l := d[k]-2 \\
\text{(Procedural Statement)} \\
\text{SKELETONIZE(PATTERN,1,MAXROW,d[l[N])} \\
\text{(Imperative Statement)} \\
l := 0; \\
d[k] := 0; \\
\text{k := 0;} \\
l := l + 1; \\
\text{j := 0;} \\
k := k + 1; \\
j := 2; \\
\text{(Conditional Statement)} \\
\end{array}
\]

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

\[
\begin{array}{c}
\text{(Path)} \\
\text{(Transition)} \\
\text{initialize pass no.} \\
\text{more dark points} \\
\text{no more dark points} \\
\text{try left/right scan} \\
\text{try top/bottom scan} \\
\text{suspend scan} \\
\text{continue scanning} \\
\text{stop scanning} \\
\text{(State)} \\
\text{1: Start} \\
\text{2: initialization of dark points} \\
\text{3: left/right scan} \\
\text{4: top/bottom scan} \\
\text{5: testing for end of scan} \\
\text{6: Exit} \\
\text{(preCondition)} \\
\text{scan permissible} \\
\text{dark pt.} \leq \text{zero} \\
\text{dark pt.} \geq \text{dark pt.} \leq \text{zero} \\
\text{dark pt.} \leq \text{zero} \\
\text{dark pt.} \geq \text{dark pt.} \leq \text{zero} \\
\text{(Procedure)} \\
\text{execute the kth scan on the entire pat.} \\
\text{(Action)} \\
\text{initialize the pass number} \\
\text{initialize d to ZERO} \\
\text{initialize the scan number} \\
\text{increment the pass number by one} \\
\text{set scan type to left/right edgpoints} \\
\text{increment the scan number by one} \\
\text{set scan type to top/bottom edgpoints} \\
\text{(postCondition)} \\
\end{array}
\]

e) Normalized infoMap: state diagram (unshaded area)
f) Normalized infoMap to be optimized: shaded areas

```plaintext
O ... x (Path)
S 0 0 0 0 0 5 (Transition)
  . 0 . more dark points
  . 0 . no more dark points
  . 0 . try left/right scan
  . o . try top/bottom scan
  . o . stop scanning
  . 0 .
L L L L L 4 (State)
  . a . 1: Start
  . d s d 2: Initialization of dark points
  . d s s 3: Left/right scan
  . d s s 4: Top/bottom scan
  . d s s 5: Testing for end of scan
  . d s s 6: Exit

[Conditional Statement]
  k := -1 to MAXSCAN
  d[k] > 0
  d[k] > d[k-2]
  d[k] = 0
  d[k] = d[k-2]

[Procedural Statement]
  SKELETONIZE(PATTERN[j,1,MAXROW,d[k]]):

[Imperative Statement]
  l := 0;
  d[k] := 0;
  k := 0;
  l := l + 1;
  j := 0;
  k := k + 1;
  j := 2;

[Conditional Statement]
```

i) Optimized and documented infoMap

```plaintext
O ... x (Path)
S 0 0 0 0 0 5 (Transition)
  . 0 . more dark points
  . 0 . no more dark points
  . 0 . try left/right scan
  . o . try top/bottom scan
  . o . stop scanning
  . 0 .
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 s s 4: Exit

[Conditional Statement]
  k := -1 to MAXSCAN
  d[k] > 0
  d[k] > d[k-2]

[Procedural Statement]
  SKELETONIZE(PATTERN[j,1,MAXROW,d[k]]):

[Imperative Statement]
  d[k] := 0;
  l := 0;
  k := 0;
  l := l + 1;
  j := 0;
  k := k + 1;
  j := 2;

[Conditional Statement]
```

```
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

1) 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".
a) Optimized and documented InfoMap: shaded area hidden in remaining figures of section 6.2.1
b) Paths for patterns execution of the SF algorithm: unshaded area

Fig. 6.2.1.1 Verification of the Straight Forward search algorithm
### Transition

<table>
<thead>
<tr>
<th>O O O O O O S S 6</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>path id.</td>
</tr>
<tr>
<td>A A</td>
<td>Run</td>
</tr>
<tr>
<td>v v</td>
<td>pattern &quot;in&quot; run</td>
</tr>
<tr>
<td>S S S S S S S S</td>
<td>Transition</td>
</tr>
<tr>
<td>1</td>
<td>initialize guess ptr. for pat within the text</td>
</tr>
<tr>
<td>2</td>
<td>location within the text</td>
</tr>
<tr>
<td>0</td>
<td>location outside the text</td>
</tr>
<tr>
<td>2</td>
<td>all pat chars matched</td>
</tr>
<tr>
<td>3</td>
<td>match at current text location</td>
</tr>
<tr>
<td>4</td>
<td>pat elem does not match with text elem</td>
</tr>
<tr>
<td>8</td>
<td>Action</td>
</tr>
<tr>
<td>1</td>
<td>set guess ptr. to zero</td>
</tr>
<tr>
<td>2</td>
<td>increment guess ptr. by 1</td>
</tr>
<tr>
<td>3</td>
<td>set text ptr. to guess ptr. value</td>
</tr>
<tr>
<td>4</td>
<td>set pat ptr. to 1</td>
</tr>
<tr>
<td>5</td>
<td>pattern not found</td>
</tr>
<tr>
<td>6</td>
<td>pattern found at guess ptr. value</td>
</tr>
<tr>
<td>7</td>
<td>increment text ptr. by 1</td>
</tr>
<tr>
<td>8</td>
<td>increment pat ptr. by 1</td>
</tr>
<tr>
<td>5 (data-object - INPUT)</td>
<td>V V</td>
</tr>
<tr>
<td></td>
<td>pattern: &quot;in&quot;</td>
</tr>
<tr>
<td></td>
<td>text: &quot;InfoMap&quot;</td>
</tr>
<tr>
<td></td>
<td>length of pattern substring</td>
</tr>
<tr>
<td></td>
<td>length of text string</td>
</tr>
<tr>
<td></td>
<td>guess ptr. for pat within the text</td>
</tr>
<tr>
<td></td>
<td>text ptr.</td>
</tr>
<tr>
<td></td>
<td>pat ptr.</td>
</tr>
<tr>
<td>5 (data-object - OUTPUT)</td>
<td>V V</td>
</tr>
<tr>
<td></td>
<td>length of pattern substring</td>
</tr>
<tr>
<td></td>
<td>length of text string</td>
</tr>
<tr>
<td></td>
<td>guess ptr. for pat within the text</td>
</tr>
<tr>
<td></td>
<td>text ptr.</td>
</tr>
<tr>
<td></td>
<td>pat ptr.</td>
</tr>
</tbody>
</table>

*Fig. 6.2.1.2* path verification for SF algorithm: pattern "in" in text "InfoMap"
Fig. 8.2.1.3 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 discussed 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


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


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.

<table>
<thead>
<tr>
<th>![Syntax Table]</th>
</tr>
</thead>
</table>

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.

(a) Pascal-type code

```
1: i := 1;
2: repeat
   3: j := 1;
   4: while j <= n
       5: and (x[i, j] = 0) do
           6: j := j + 1;
       7: t := t + 1;
   8: until (i > n) or (j > n);
9: if j > n
    10: then writeln
        11: ('The...row is', i, '.');
```

(b) Pascal code rewritten as infoMap

```
  s
  d s d
data s   d s
  d s
  d s
  d s
  d s
  f i
  f t
  f j
  s = 1
  j = 1
  j = j + 1,
  i = i + 1.
  print ('The...row is', i, ');
```

(c) Pancode

```
i := 1
do checkrow: if j <= n
    j := 1
    do checknumber: if j <= n
        if x[i, j] = 0
            j := j + 1
    end
  end
end
end
```

(d) Pancode rewritten as infoMap

```
L, L, L, L, L, L, [Node]
  s
  d s d
data s   d s
  d s
  d s
  d s
  d s
  f i
  f t
  f j
  s = 1
  j = 1
  j = j + 1,
  i = i + 1.
  print ('The...row is', i, ');
```

Fig. 4.3.2 All-zero row Program
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.
j := 1; i := FAIL(1) := 0;

while(j < m) do
  /* FAIL(1) ... FAIL(j) is known - compute FAIL(j+1) */
  while(i > 0 and PAT(j) <> PAT(i)) do
    i := FAIL(i); end;
  i := i+1; j := j+1;
  if(PAT(j) = PAT(i)) then
    FAIL(j) := FAIL(i)
  else
    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)
c) Normalized InfoMap: shaded areas to be filled in

\[
\begin{align*}
\text{(Path)} & \quad x \quad \text{(transition)} \\
\text{(State)} & \quad L \quad L \quad L \quad L \quad L \quad L \quad 5 \\
\text{(Conditional Statement)} & \quad \text{PreCondition} \\
& \quad | < m \quad \quad I \quad I \quad \text{more elements left} \\
& \quad I > 0 \quad \quad L \quad c \quad \text{left pointer greater than zero} \\
& \quad \text{PAT}(i) == \text{PAT}(i) \quad \quad L \quad c \quad \text{element } j \text{ not equal to element } i \\
& \quad \text{PAT}(i) = \text{PAT}(i) \quad \quad L \quad I \quad \text{element } j \text{ equals element } i \\
\text{(Imperative Statement)} & \quad \text{Action} \\
& \quad I := 1; \quad 1 \quad \text{set right pointer to 1} \\
& \quad I := 0; \quad 2 \quad \text{set left pointer to zero} \\
& \quad \text{FAIL}(1) := 0; \quad 3 \quad \text{set first element of Fail to zero} \\
& \quad I := \text{FAIL}(i); \quad 1 \quad \text{calculate new left pointer} \\
& \quad I := I+1; \quad 1 \quad \text{increment left pointer by 1} \\
& \quad I := I+1; \quad 2 \quad \text{increment right pointer by 1} \\
& \quad \text{FAIL}(i) := \text{FAIL}(i); \quad 1 \quad \text{set } j \text{ elem. of Fail to } i \text{ elem. of Fail} \\
& \quad \text{FAIL}(i) := I; \quad 1 \quad \text{set } j \text{ elem. of Fail to left pointer} \\
\end{align*}
\]

\[
\begin{align*}
\text{(Conditional Statement)} & \quad \text{PostCondition} \\
& \quad \text{O G G G G G G O O} \\
\text{(Path)} & \quad x \quad \text{(transition)} \\
\text{(State)} & \quad L \quad L \quad L \quad L \quad L \quad L \quad L \quad 5 \\
\text{(Conditional Statement)} & \quad \text{PreCondition} \\
& \quad | < m \quad \quad I \quad I \quad \text{more elements left} \\
& \quad I > 0 \quad \quad L \quad c \quad \text{left pointer greater than zero} \\
& \quad \text{PAT}(i) == \text{PAT}(i) \quad \quad L \quad c \quad \text{element } j \text{ not equal to element } i \\
& \quad \text{PAT}(i) = \text{PAT}(i) \quad \quad L \quad I \quad \text{element } j \text{ equals element } i \\
\text{(Imperative Statement)} & \quad \text{Action} \\
& \quad I := 1; \quad 1 \quad \text{set right pointer to 1} \\
& \quad I := 0; \quad 2 \quad \text{set left pointer to zero} \\
& \quad \text{FAIL}(1) := 0; \quad 3 \quad \text{set first element of Fail to zero} \\
& \quad I := \text{FAIL}(i); \quad 1 \quad \text{calculate new left pointer} \\
& \quad I := I+1; \quad 1 \quad \text{increment left pointer by 1} \\
& \quad I := I+1; \quad 2 \quad \text{increment right pointer by 1} \\
& \quad \text{FAIL}(i) := \text{FAIL}(i); \quad 1 \quad \text{set } j \text{ elem. of Fail to } i \text{ elem. of Fail} \\
& \quad \text{FAIL}(i) := I; \quad 1 \quad \text{set } j \text{ elem. of Fail to left pointer} \\
\end{align*}
\]

d) Normalized InfoMap: states and transitions added (shaded areas)
e) Normalized InfoMap: state diagram (unshaded area)

f) Normalized InfoMap to be optimised: shaded area
g) Optimized and documented infoMap

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

2.: evaluation of Remaining elements
   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.: \textit{start initialization of pointers}
   \begin{itemize}
   \item initialize pointers, \texttt{CONTINUE AT 2}
   \item \( j := 1; i := 0; \FAIL(1) := 0; \texttt{goto 2} \)
   \end{itemize}

2.: \textit{evaluation of Remaining elements}
   \begin{itemize}
   \item no more elements left, \texttt{CONTINUE AT 4}
   \item \( j \geq m \rightarrow \texttt{goto 4} \)
   \end{itemize}
   
   \begin{itemize}
   \item adjacent characters are different, \texttt{CONTINUE AT 3}
   \item \(< m \text{ and } i > 0 \text{ and } \texttt{PAT}(j) \neq \texttt{PAT}(i) \rightarrow i := \FAIL(i); \texttt{goto 3} \)
   \end{itemize}

   \begin{itemize}
   \item update pointers, \texttt{CONTINUE AT 3}
   \item \(< m \text{ and } (i < 0 \text{ or } \texttt{PAT}(j) = \texttt{PAT}(i)) \rightarrow i := i + 1; j := j + 1; \texttt{goto 3} \)
   \end{itemize}

3.: \textit{updating Fail array}
   \begin{itemize}
   \item element \( j \) equals element \( i \), \texttt{CONTINUE AT 2}
   \item \( \texttt{PAT}(j) = \texttt{PAT}(i) \rightarrow \FAIL(j) := \FAIL(i); \texttt{goto 2} \)
   \end{itemize}

   \begin{itemize}
   \item element \( j \) not equal to element \( i \), \texttt{CONTINUE AT 2}
   \item \( \texttt{PAT}(j) \neq \texttt{PAT}(i) \rightarrow \FAIL(j) := i; \texttt{goto 2} \)
   \end{itemize}

4.: \textit{Exit}

\begin{itemize}
\item 1) source code (generated)
\end{itemize}

\textbf{Fig. 5.1.12 Control flow of Computing table for KMP pattern matching}
1. \[ j := k := 1; \]
2. \[ \textbf{while}(k <= n) \textbf{do} \]
3. \[ \quad \textbf{while}(j > 0 \text{ and } \text{TXT}(k) \neq \text{PAT}(j)) \textbf{do} \]
4. \[ \quad \quad j := \text{FAIL}(j); \]
5. \[ \quad \textbf{end}; \]
6. \[ \quad \textbf{if}(j = m) \textbf{then} \]
7. \[ \quad \quad \textbf{return}("pattern found at", k - m) \]
8. \[ \quad \textbf{else do} \]
9. \[ \quad \quad k := k+1; \]
10. \[ \quad \quad j := j+1; \]
11. \[ \quad \textbf{end}; \]
12. \[ \textbf{return}("pattern not found"); \]

\[ \textbf{a) source code: original} \]

1: \[ j := k := 1; \]
2: \[ \textbf{while}(k <= n) \textbf{do} \]
3: \[ \quad \textbf{while}(j > 0 \text{ and } \text{TXT}(k) \neq \text{PAT}(j)) \textbf{do} \]
4: \[ \quad \quad j := \text{FAIL}(j); \]
5: \[ \quad \textbf{end}; \]
6: \[ \quad \textbf{if}(j = m) \textbf{then} \]
7: \[ \quad \quad \textbf{return}("pattern found at", k - m) \]
8: \[ \quad \textbf{else do} \]
9: \[ \quad \quad k := k+1; \]
10: \[ \quad \quad j := j+1; \]
11: \[ \quad \textbf{end}; \]
12: \[ \textbf{return}("pattern not found"); \]

\[ \textbf{b) edited source code: line nos removed, states 1 .. 5 identified} \]

Note: Nos. 1 .. 9 in a) are line numbers of original algorithm and should not be confused with states 1 .. 5 in b).
c) Normalized infoMap: shaded areas to be filled in

Note: "c and c" means all other conditions besides "t and t"
O . . . . . . x [Path]
S 0 0 0 0 0 0 6 [Transition]
  . 0 . . . . compute Fail table and initialize ptrs.
  . 0 . . . . enough text left
  . 0 . . . . not enough left
  . . 0 . . . match at current text location
  . . . 0 . . all pat chars matched
  . . . . 0 . calculate shift for new text location
  . . . . . 4 [State]
  . . . . . . 1: Start
  . d s s s . d 2: Evaluating remaining text
  . . d l s s 3: Matching text elem with pat elem
  . . . . . . 4: Exit

[Conditional Statement]
  . 0 0 0 0 3 [preCondition]
k <= n
TXT(k) <> PAT(j)
j = m

[Procedural Statement]
  . S S S S S S [Procedure]
FAIL()

[Imperative Statement]
  . S S S S S S [Action]
j := 1;
1: k := 1;
2: return("pattern not found");
3: return failure message
k := k+1;
4: increment text pointer
j := j+1;
5: increment pat pointer
return("pattern found at", k-m);
6: return success message
j := FAIL(j);
7: obtain value of shift

[Conditional Statement]
  . 0 0 0 0 0 1 [postCondition]

\( \odot \) 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
for every character c in the input alphabet do \text{DELT}A1(c) := m;

for j := m to 1 by -1 do /* compute DELT2, initialize DELT2 */
  if(DELT1(PAT(j)) = m) then DELT1(PAT(j)) = m-j;
  DELT2(j) := 2*m-j;
end;

/* compute DELT2 */

j := m; t := m+1;

while(j > 0) do
  f(j) := t;
  while(t <= m and PAT(j) <> PAT(t)) do
    DELT2(t) := min(DELT2(t), m-j);
    t := f(t);
  end;
  j := j-1;
  t := t-1;
end;

for k := 1 to t do DELT2(k) := min(DELT2(k), m+t-k);

/* The following steps were added by Mehlhorn [4] to ensure */
/* correct values for DELT2 in all cases */

tp := f(t);

while(t <= m) do
  while(t <= tp) do
    DELT2(t) := min(DELT2(t), tp-t+m);
    t := t+1;
  end;
  tp := f(tp);
end;

\text{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;
4:   DELTA2(j) := 2^m-j;
5:   /* compute DELTA2 */
6:   j := m; t := m+1;
7:   while(j > 0) do
8:     f(j) := t;
9:     while (t <= m and PAT(j) <> PAT(t)) do
10:        DELTA2(t) := min(DELTA2(t), m-j);
11:        t := f(t);
12:     j := j-1;
13:     t := t-1;
14:   end;
15:   for k := 1 to t do DELTA2(k) := min(DELTA2(k), m+t-k);
16: /* The following steps were added by Mehlhorn [4] to ensure */
17: /* correct values for DELTA2 in all cases */
18:   tp := f(t);
19:   while (t <= m) do
20:      while (t <= tp) do
21:         DELTA2(t) := min(DELTA2(t), tp-t+m);
22:         t := t+1;
23:      end;
24:      tp := f(tp);
25:   end;

b) edited source code: line nos removed, states 1 .. 9 identified
[Conditional Statement]

1. \( \leq m \) to 1 by -1
2. \( \text{DELTA}(\text{PAT})[j] \Rightarrow m \)
3. \( i > 0 \)
4. \( t \Leftarrow m \)
5. \( \text{PAT} \Leftarrow \text{PAT}[j] \)
6. \( i > t \) to 1
7. \( t \Leftarrow t' \)

[Loop Statement]

1. \( \text{DELTA}(\text{PAT})[j] \Rightarrow m+1 \)
2. \( t > m+1 \)
3. \( i > t \)
4. \( \text{DELTA}(\text{PAT})[j] \Rightarrow \min(\text{DELTA}(\text{PAT}), m+1) \)
5. \( t > t' \)
6. \( i > t \)
7. \( \text{DELTA}(\text{PAT})[j] \Rightarrow \min(\text{DELTA}(\text{PAT}), t-1) \)
8. \( t > t' \)

[Conditional Statement]

1. \( \leq m \) to 1 by -1
2. \( \text{DELTA}(\text{PAT})[j] \Rightarrow m \)
3. \( i > 0 \)
4. \( t \Leftarrow m \)
5. \( \text{PAT} \Leftarrow \text{PAT}[j] \)
6. \( i > t \) to 1
7. \( t \Leftarrow t' \)
initialize DELTA1 array
no more input alphabet
and computing DELTA1 & initializing DELTA2
more elem. to process
compute DELTA1
initialize DELTA2
and DELTA2 computation
calculate restart pg
calculate min DELTA2 shift
end calculating DELTA2 shift
end calculating new DELTA2 shift
and verification of shift
more shift elem. to verify
verify minimum shift
calculate new restart pg
start initialization of DELTA1 array
initialization of DELTA2 array
computing DELTA1
testing for end of DELTA2 computation
comparing adjacent array elem.
computing minimum values for DELTA2
verifying end of DELTA2 verification
verifying minimum DELTA2 elem.
exit

for every character c in the input alphabet
1st elem <= pattern length
elem. of DELTA1 <= length of length elem. > zero
elem <= pattern length
pat elem. j not equal to pat elem t
f <= current elem. <= f
right ptr. <= restart ptr.
set DELTA1 elem. to pat length
initialize DELTA2
increment match ptr
calculate minimum DELTA2 shift
calculate next t
calculate new DELTA2 shift
calculate new restart ptr
verify minimum DELTA2 shift
increment right ptr. t

debug condition
} Normalized inf/sup to be optimized; shaded areas
Optimized and documented infolayout

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.
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;
7. end;
8. if(j = 0) then
9.   return("pattern found at", k+1)
10. else /* shift the pattern */
11.     k := k+max(DELA1(TXT[k]), DELTA2[j]);
12. end;
13. return("pattern not found");

a) source code: original

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;
7: end;
8: if(j = 0) then
9:   return("pattern found at", k+1)
10: else /* shift the pattern */
11:     k := k+max(DELA1(TXT[k]), DELTA2[j]);
12: end;
13: return("pattern not found");

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)
(Procedural Statement)
\[
\begin{align*}
&k = m; \\
&j = m; \\
&\text{return(\"pattern not found\");} \\
&j = j - 1; \\
&k = k - 1;
\end{align*}
\]

(Imperative Statement)
\[
\begin{align*}
k &= m; \\
j &= m; \\
\text{return(\"pattern not found\");} \\
j &= j - 1; \\
k &= k - 1;
\end{align*}
\]

(Conditional Statement)
\[
\begin{align*}
k &= n; \\
j &= 0; \\
\text{TXT}(k) &= \text{PAT}(j); \\
j &= 0; \\
k &= k - 1;
\end{align*}
\]

(Procedural Statement)
\[
\begin{align*}
k &= m; \\
j &= m; \\
\text{return(\"pattern not found\");} \\
j &= j - 1; \\
k &= k - 1;
\end{align*}
\]

(Imperative Statement)
\[
\begin{align*}
k &= m; \\
j &= m; \\
\text{return(\"pattern not found\");} \\
j &= j - 1; \\
k &= k - 1;
\end{align*}
\]

(Conditional Statement)
\[
\begin{align*}
k &= n; \\
j &= 0; \\
\text{TXT}(k) &= \text{PAT}(j); \\
j &= 0; \\
k &= k - 1;
\end{align*}
\]

(Procedural Statement)
\[
\begin{align*}
k &= m; \\
j &= m; \\
\text{return(\"pattern not found\");} \\
j &= j - 1; \\
k &= k - 1;
\end{align*}
\]

(Imperative Statement)
\[
\begin{align*}
k &= m; \\
j &= m; \\
\text{return(\"pattern not found\");} \\
j &= j - 1; \\
k &= k - 1;
\end{align*}
\]

(Conditional Statement)
\[
\begin{align*}
k &= n; \\
j &= 0; \\
\text{TXT}(k) &= \text{PAT}(j); \\
j &= 0; \\
k &= k - 1;
\end{align*}
\]

(Procedural Statement)
\[
\begin{align*}
k &= m; \\
j &= m; \\
\text{return(\"pattern not found\");} \\
j &= j - 1; \\
k &= k - 1;
\end{align*}
\]

(Imperative Statement)
\[
\begin{align*}
k &= m; \\
j &= m; \\
\text{return(\"pattern not found\");} \\
j &= j - 1; \\
k &= k - 1;
\end{align*}
\]
g) Optimized and documented infoMap
1. Start
   compute shift table and initialize text ptr., CONTINUE at 2
   DELTA1=DELTA20; 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
   all pat chars matched, CONTINUE at 4
   calculate shift for new text location, CONTINUE at 2
   TXT(k) = PAT(j) and j<= 0 -> j:= j-1; k:= k-1; goto 3
   TXT(k) = PAT(j) and j> 0 -> Return (pattern found at', k+1); goto 4

4. Exit
   1) 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
c) Normalized infoMap: shaded area to be fill in

\[
\begin{array}{c}
\begin{array}{c}
0 \ldots x [Path] \\
S 0 0 0 0 4 [Transition] \\
. 0 \ldots \\
. 0 \ldots \\
. 0 \ldots \\
. 0 \ldots \\
. 0 \ldots \\
.\ldots \ldots \\
L L L L 3 [State] \\
. 1 \ldots \\
. d l s \\
.\ldots \ldots \\
\end{array}
\end{array}
\]

\textbf{[Conditional Statment]}
\begin{array}{c}
. G G G G 2 [preCondition] \\
l := 0 to ASIZE \\
p := psstr; *p; p++ \\
. t f \ldots \\
. t f \ldots \\
.\ldots \ldots \\
.\ldots \ldots \\
\end{array}

\textbf{[Imperative Statment]}
\begin{array}{c}
. S S S S 2 [Action] \\
TD1[i] := Plan+1; \\
TD1[*p] := Plan - (p - psstr); \\
. 1 \ldots \\
. 1 \ldots \\
.\ldots \ldots \\
.\ldots \ldots \\
\end{array}

\textbf{[Conditional Statment]}
\begin{array}{c}
. G G G G [postCondition] \\
\end{array}

\]

\]

\[
\begin{array}{c}
0 \ldots x [Path] \\
S 0 0 0 0 4 [Transition] \\
. 0 \ldots \\
. 0 \ldots \\
. 0 \ldots \\
. 0 \ldots \\
. 0 \ldots \\
.\ldots \ldots \\
L L L L 3 [State] \\
. 1 \ldots \\
. d l s \\
.\ldots \ldots \\
\end{array}
\]

\textbf{[Conditional Statment]}
\begin{array}{c}
. G G G G 2 [preCondition] \\
l := 0 to ASIZE \\
p := psstr; *p; p++ \\
. t f \ldots \\
. t f \ldots \\
.\ldots \ldots \\
.\ldots \ldots \\
\end{array}

\textbf{[Imperative Statment]}
\begin{array}{c}
. S S S S 2 [Action] \\
TD1[i] := Plan+1; \\
TD1[*p] := Plan - (p - psstr); \\
. 1 \ldots \\
. 1 \ldots \\
.\ldots \ldots \\
.\ldots \ldots \\
\end{array}

\textbf{[Conditional Statment]}
\begin{array}{c}
. G G G G [postCondition] \\
\end{array}

\]

\[
\]

d) Normalized infoMap: states and transitions added (shaded areas)
### e) Normalized infoMap: state diagram (unshaded area)

<table>
<thead>
<tr>
<th>Path</th>
<th>Transition</th>
<th>State</th>
<th>PreCondition</th>
<th>Action</th>
<th>PostCondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>S O O O O 4</td>
<td>(Conditional Statement)</td>
<td>G G G G 2</td>
<td>l := 0 to ASIZE</td>
<td>TD1[1] := Plan + 1;</td>
<td>G G G G</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>l f</td>
<td>TD1[p] := Plan - (p - pstr);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>elem, within range</td>
<td>fill in values from pattern string</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>p := pstr; p++;</td>
<td>T L L L 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>t f</td>
<td>1: Start initialization of table</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fill in values</td>
<td>2: Filling in of values into table</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>end fill in of values</td>
<td>3: Exit</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Path</th>
<th>Transition</th>
<th>State</th>
<th>PreCondition</th>
<th>Action</th>
<th>PostCondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>S O O O O 4</td>
<td>(Conditional Statement)</td>
<td>G G G G 2</td>
<td>l := 0 to ASIZE</td>
<td>TD1[1] := Plan + 1;</td>
<td>G G G G</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>l f</td>
<td>TD1[p] := Plan - (p - pstr);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>elem, within range</td>
<td>fill in values from pattern string</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>p := pstr; p++;</td>
<td>T L L L 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>t f</td>
<td>1: Start initialization of table</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fill in values</td>
<td>2: Filling in of values into table</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>end fill in of values</td>
<td>3: Exit</td>
<td></td>
</tr>
</tbody>
</table>

- O...O x (Path)
- S...S 4 (Transition)
- T L L L 3 (State)
- G...G 2 (PreCondition)
- G...G (PostCondition)
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 Statement] . G G . (preCondition)
[Imperative Statement] . 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
  i:= 0 to ASIZE . t . . elem. within range
  p:=pstr; *p; p++ . 1 . . 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}
      l := ([j]-lshift);
      if(i<0) or (p[l][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;

\(1:\) 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}
      l := ([j]-lshift);
      if(i<0) or (p[l][j] <> p[i])
        then gotshift := true
      else begin  \{get next matching shift\}
        lshift := lshift + 1;
        lshift := matchshift(j,lshift)
      end;
    end;
end;
TD2[j] := lshift;  \{set final shift\}
end;

\(5:\) edited source code: states 1 . . . 6 identified
c) Normalized infoMap: shaded areas to be filled in

\[ \begin{align*}
0 & \quad \text{(Path)} \\
0 & \quad \text{(Transition)}
\end{align*} \]

\[ \begin{align*}
\text{(Conditional Statement)} & \quad \text{(preCondition)} \\
\text{(Imperative Statement)} & \quad \text{(action)} \\
\text{(Conditional Statement)} & \quad \text{(postCondition)}
\end{align*} \]

\[ \begin{align*}
\text{(Conditional Statement)} & \quad \text{(preCondition)} \\
\text{(Imperative Statement)} & \quad \text{(action)} \\
\text{(Conditional Statement)}
\end{align*} \]

\[ \begin{align*}
\text{(Conditional Statement)} & \quad \text{(preCondition)} \\
\text{(Imperative Statement)} & \quad \text{(action)} \\
\text{(Conditional Statement)}
\end{align*} \]

\[ \begin{align*}
\text{(Conditional Statement)} & \quad \text{(preCondition)} \\
\text{(Imperative Statement)} & \quad \text{(action)} \\
\text{(Conditional Statement)}
\end{align*} \]

\[ \begin{align*}
\text{c) Normalized infoMap: shaded areas to be filled in}
\end{align*} \]

\[ \begin{align*}
\text{d) Normalized infoMap: states and transitions added (shaded areas)}
\end{align*} \]
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;  \(\{\text{j scans the ordered pattern}\}\)
    while(j<m) and (p[I][j] = text[k+I[j]])
        do j:= j+1;
    if(j=m)  \(\{\text{all pattern chars matched}\}\)
        then gotmatch := true
        else begin
            \(\{\text{shift pattern}\}\)
            delta1 := TD1[text[k+m]];
            delta2 := TD2[j];
            k := k + max(delta1, delta2);
        end;
end;
if(gotmatch = true)
then Search := k  \(\{\text{pattern match found at text location k}\}\)
else Search := (-1)  \(\{\text{no pattern match found in text}\}\)

a) source code: original

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

b) edited source code: states 1 . . . 6 identified
c) Normalized infoMap: shaded areas to be filled in
d) Normalized infoMap: states and transitions added (shaded areas)
1) Normalized infoMap to be optimized: shaded areas
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;  
  while(i<m) and (p[i] = text[k+i])
    do i:=i+1;
  if(i=m)  
    then gotmatch := true
    else k := k+TD1[text[k+m]][shift pattern]
end;
if(gotmatch = true)  
  then QSearch:=k  
  else QSearch:=-1  

1 :  (Quick Search for a string in text)
    gotmatch:=false;
    k:=0;
2 :  while(gotmatch = false) and (k+m <= n)
     do begin
       i:=0;  
       while(i<m) and (p[i] = text[k+i])
         do i:=i+1;
       if(i=m)  
         then gotmatch := true
         else k := k+TD1[text[k+m]][shift pattern]
     end;
3 :  if(gotmatch = true)  
     then QSearch:=k  
     else QSearch:=-1  

   b) edited source code: states 1 . . . 6 identified
c) Normalized InfoMap: shaded areas to be filled in
d) Normalized infoMap: states and transitions added (shaded areas)
1) Normalized infoMap to be optimized: shaded area

O  .  .  .  .  .  .  .  .  .  x  [Path]
S 0 0 0 0 0 0 0 0 0 6  [Transition]
   0  .  .  .  .  .  .  .  .  .  compute shift table and initialize text ptr.
   0  .  .  .  .  .  .  .  .  .  location within the text
   0  .  .  .  .  .  .  .  .  .  location outside the text
   0  .  .  .  .  .  .  .  .  .  match at current text location
   0  .  .  .  .  .  .  .  .  .  matching suspended
   0  .  .  .  .  .  .  .  .  .  all pat chars compared
   0  .  .  .  .  .  .  .  .  .  calculate new text location
   0  .  .  .  .  .  .  .  .  .  pattern match occur
   0  .  .  .  .  .  .  .  .  .  no pattern match occur
L L L L L L L L L 1 6  [State]
   1  .  .  .  .  .  .  .  .  .  1: Start
   0  .  .  .  .  .  .  .  .  .  2: Evaluating remaining text
   0  .  .  .  .  .  .  .  .  .  3: Matching pat elem with text elem
   0  .  .  .  .  .  .  .  .  .  4: End of pattern testing
   0  .  .  .  .  .  .  .  .  .  5: Testing for pattern occurrence
   0  .  .  .  .  .  .  .  .  .  6: Exit

(Conditional Statement)
   gofmatch = false  G 0 0 0 0 0 0 0 0 5  [preCondition]
   kmn <= m  .  .  .  .  .  .  t  c  i  t  no match
   p[i] = text[k+i] .  .  .  .  .  .  1  c  .  .  .  location within text
   km  .  .  .  .  .  .  .  .  .  .  1  c  .  .  .  pat elem equals text elem
   lm  .  .  .  .  .  .  .  .  .  .  .  .  .  location outside text
   all pat chars compared

(Procedural Statement)
   S S S S S S S S S 1  [Procedure]
   1  .  .  .  .  .  .  .  .  .  compute table TD[i][j]

(Imperative Statement)
   S S S S S S S S S 8  [Action]
   gofmatch = false  .  .  .  .  .  .  2  .  set gofmatch to false
   k = 0  .  .  .  .  .  .  .  3  .  set text pointer to zero
   i = 0  .  .  .  .  .  .  .  4  .  set scan pointer to zero
   gofmatch = true  .  .  .  .  .  .  5  .  set gofmatch to true
   i = i + 1  .  .  .  .  .  .  .  6  .  increment scan pointer by 1
   QSearch := -1  .  .  .  .  .  .  .  7  .  no substring match found in text
   QSearch := k  .  .  .  .  .  .  .  .  8  .  substring match found at text[k]
   k := k + TD[i][text[k+m]]  .  .  .  .  .  .  9  .  calculate new location

(Conditional Statement)
   G 0 0 0 0 0 0 0 0 9  [postCondition]

2) 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 -> l:= 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] -> l:= 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[l] = text[k+1] -> k:= k+TD1[text[k+m]]; goto 2

4.: Exit
   !) 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\_dcest() 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 \( e_{vw} \neq e_{ij}. \)
8. If \( w[e_{vw}] < w[e_{ij}] \) and \( \text{degree}[v] < b \) and \( \text{degree}[w] < b \)
   then \( S = S - \{e_{ij}\} \cup \{e_{vw}\}, \) goto (10).
9. If \( \text{degree}[v] < b \) and \( \text{degree}[w] < b \) then goto (10).
   end {loop (6)}
10. end {loop (2)}
11. Return \( S \) and its cost.
(Declaration) 
<table>
<thead>
<tr>
<th>Declaration</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>cost of S</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td></td>
</tr>
<tr>
<td>v</td>
<td></td>
</tr>
<tr>
<td>d[i]</td>
<td></td>
</tr>
<tr>
<td>DCST</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Conditional Statement) 
<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>F F F F F F F F F F F F F F F F</td>
<td>loop n times</td>
</tr>
<tr>
<td>loop &lt; n</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>F F F F F F F F F F F F F F F F</td>
<td>loop 20 times</td>
</tr>
<tr>
<td>loop &lt;= 20</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>W &gt; W[i][j]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>d[v] &lt; b</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>d[w] &lt; b</td>
<td></td>
</tr>
</tbody>
</table>

(Procedural Statement) 
<table>
<thead>
<tr>
<th>Statement</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>F F F F F F F F F F F F F F F F</td>
<td>generate DCST</td>
</tr>
<tr>
<td>F F F F F F F F F F F F F F F F</td>
<td>Return S and its cost</td>
</tr>
<tr>
<td>F F F F F F F F F F F F F F F F</td>
<td>let eij be a random edge in S</td>
</tr>
<tr>
<td>F F F F F F F F F F F F F F F F</td>
<td>let Ti &amp; Tj be the subtrees</td>
</tr>
<tr>
<td>F F F F F F F F F F F F F F F F</td>
<td>reachable from i and j</td>
</tr>
<tr>
<td>F F F F F F F F F F F F F F F F</td>
<td>let evw be a random &amp; not equal to eij</td>
</tr>
<tr>
<td>F F F F F F F F F F F F F F F F</td>
<td>update set of edges in DCST</td>
</tr>
</tbody>
</table>

(Conditional Statement) 
<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>F F F F F F F F F F F F F F F F</td>
<td>postCondition</td>
</tr>
</tbody>
</table>

Fig. 5.2.3.4 Data flow for Primal algorithm
a) source code: original (see page 137)

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

Definitions: \( S \) : set of edges already in DCST. \( S \subseteq E. \)
\[ \forall i \in V, \text{ degree}[i] \text{ is the current degree } \]
\[ \text{of } i \text{ in DCST}. \]

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 \( \neg \exists e \in E \text{ s.t. } e \neq e_{ij} \text{ and } e \text{ connects} \)
   \( T_i \) and \( T_j \), goto 2.
5:   loop 20 times
      begin
      Let \( e_{vw} \) be a random edge in \( E \) and \( e_{uv} \neq e_{ij}. \)
6:      If \( w[e_{vw}] < w[e_{ij}] \text{ and degree}[v] < b \text{ and degree}[w] < b \)
      then \( S = S - \{ e_{ij} \} \cup \{ e_{vw} \}, \text{ goto 2.} \)
7:      If \( \text{degree}[v] < b \text{ and degree}[w] < b \text{ then goto 2.} \)
   end \{loop 5\}
   end \{loop 2\}
8:   Return \( S \) and its cost.

b) edited source code: line nos removed, states 1..8 identified
c) Normalize $\psi$; map; shaded area is to be filled in
O (Path)
S o o o o o o o o o 13 (Transition)
  o
  o
  o
  o
  o
  o
  o
  o
  o
  L L L L L L L L L L L B (State)
s d s d d d d d
d d s d d d d d
d d d d d d d d
d d d d d d d d
d d d d d d d d
d d d d d d d d
d d d d d d d d
d d d d d d d d
(Conditional Statement)
loop <= n
  t f
  f f
loop <= 20
  t f
  f c
  t c
W[evw] < W[ij]
d[v] < b
d[w] < b
d[v] < b
d[w] < b
(Procedural Statement)
S S S S S S S S S S S S S S S S 1 (Procedure)
1 (Action)
  S S S S S S S S S S S S S S 5
  1
  1
  1
S := S - [e;j] U [evw]
(Conditional Statement)
G G G G G G G G G G G G G G G G
(d) Normalized InfoMap: states and transitions added (shaded area)
DCST generated
no more nodes to examine
more nodes to examine
edge \( ij \) removed
edge \( ij \) not removed
a different edge from \( ei \) exists
no different edge from \( ei \) exist
no more random edge to generate
more random edge to generate
update set of edges in DCST
no update to set of edges in DCST
degree of nodes \( v \) & \( w \) less than \( b \)
degree of \( v \) & \( w \) not less than \( b \)

1: Start generation of DCST
2: return of \( S \) and its cost
3: identifying subtrees
4: testing of edges
5: selecting a random edge
6: updating set of edges in DCST
7: testing degree of node
8: Exit

loop \( \times n \) times
loop \( 20 \) times
\( e \leftarrow e_i \) and \( e \) connects \( T_i \) & \( T_j \)
weight of random edge \( v \leftarrow \) edge \( i \)
degree of node \( v \) less than \( b \)
degree of \( v \) less than \( b \)
degree of node \( v \) less than \( b \)
degree of node \( v \) less than \( b \)

generate DCST
Return \( S \) and its cost
let \( e_i \) be a random edge in \( S \)
let \( T_i \) & \( T_j \) be the subtrees
reachable from \( i \) and \( j \)
let \( e_i w \) be a random & not equal to \( e_i \)

update set of edges in DCST

*) Normalised InfoMap to be optimised: shaded area
g) Optimized and documented InputMap

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

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

Algorithm: (1) Let \( P = \{1\}, S = \emptyset \).
\[ \forall j \in V, \text{ let ini}[j] = 1, u[j] = w_{1j}. \]
(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
\[ \forall j \in V - P, \text{ if } w_{kj} < u[j], \text{ let } u[j] = w_{kj}, \text{ ini}[j] = k. \]
end
(6) For each \( i \in V \) s.t. \( \text{degree}[i] > b \) do
(7) While \( \text{degree}[i] > b \) 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 \( D = \{(x, y) \mid x \in T_i \text{ and } y \in T_j \text{ and} \)
\[ \{ \text{degree}[x] > b - 1 \text{ and } x \neq i, \text{ or} \]
\[ \text{degree}[j] > b - 1 \text{ and } y \neq j, \text{ or} \]
\[ \text{degree}[j] = 1 \text{ and } j \neq y, \text{ or} \]
\[ e_{xy} \notin E, \text{ or } e_{xy} \in S, \text{ or } i = x \} \]
\[ \text{If } D = T_i \times T_j \text{ then } f[j] = \infty \]
else \[ f[j] = \max_{(x, y) \in (T_i \times T_j) - D} w_{ij} - w_{xy}. \]
end.
(13) If \( \forall j \in V, f[j] = \infty \), then error(“No feasible solution”).
(14) Let \( f[k] = \max_{j \in V, |j| < \infty} f[j] \).
(15) Let \( x \) and \( y \) be the nodes defining \( f[k] \).
(16) Let \( S = S - \{e_{ij}\} \cup \{e_{xy}\} \).
end
(18) Return \( S \) and its cost.
a) source code: original (see page 145)

1: Let $P = \{1\}$, $S = \emptyset$.
2: $\forall j \in V$, let $\text{ini}[j] = 1$, $u[j] = w_{ij}$.
3: While $P \neq V$ do 
   begin
     Let $u[k] = \min_{j \in V - P} u[j]$.
     $P = P \cup \{k\}$, $S = S \cup \{(\text{ini}[k], k), (k, \text{ini}[k])\}$.
   end
4: If $P \neq V$ then 
5:   $\forall j \in V - P$, if $w_{kj} < u[j]$, let $u[j] = w_{kj}$, $\text{ini}[j] = k$.
   end
6: For each $i \in V$ s.t. $\text{degree}[i] > b$ do 
7:   While $\text{degree}[i] > b$ do 
       begin
         For each $j \in V$ do 
         9:           If $e_{ij} \not\in S$ 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 $-\exists e \in E$ s.t. $e \neq e_{ij}$ and $e$ connects $T_i$ and $T_j$ 
                 then $f[j] = \infty$.
12:             Let $D = \{(x, y) \mid x \in T_i \text{ and } y \in T_j \text{ and}$
                 $\{ \text{degree}[x] > b - 1 \text{ and } x \neq i, \text{ or}$
                 $\text{degree}[j] > b - 1 \text{ and } y \neq j, \text{ or}$
                 $\text{degree}[j] = 1 \text{ and } j \neq y, \text{ or}$
                 $e_{xy} \not\in E, \text{ or } e_{xy} \in S, \text{ or } i = x \}$
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.
9:       end
14:     end
7:   end
6: end
4: end
3: end 

15: Return $S$ and its cost.

b) edited source code: line nos removed, states 1..15 identified
c) Normalized infomap: shaded areas to be filled in
<table>
<thead>
<tr>
<th>Conditional Statement</th>
<th>( G )</th>
<th>( G )</th>
<th>( G )</th>
<th>( G )</th>
<th>( G )</th>
<th>( G )</th>
<th>( G )</th>
<th>( G )</th>
<th>( G )</th>
<th>( G )</th>
<th>( G )</th>
<th>( G )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precondition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( P -&gt; V )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( W_i &lt; U_j )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{degree}(i) &gt; b )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{entry} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedural Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
</tr>
<tr>
<td>( 1 )</td>
</tr>
<tr>
<td>( 1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Imperative Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P &gt; 1 )</td>
</tr>
<tr>
<td>( S &gt; 0 )</td>
</tr>
<tr>
<td>( \text{init}(i) = 1 )</td>
</tr>
<tr>
<td>( U )</td>
</tr>
<tr>
<td>( S )</td>
</tr>
<tr>
<td>( 3 )</td>
</tr>
<tr>
<td>( 1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conditional Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G )</td>
</tr>
<tr>
<td>Postcondition</td>
</tr>
</tbody>
</table>

\( d \) Normalized InfoMap: states and transitions added (shaded area)
\( P \rightarrow \{ l \} \)
\( S > \{ \} \)
\( \text{init}[\cdot] \rightarrow 1 \)
\( u[\cdot] \rightarrow W[\cdot] \)
\( \cup[\cdot] = \text{min}(\cup[\cdot]) \)
\( P = P \cup \{ k \} \)
\( S > S \cup \{(\text{init}[\cdot], \{\text{init}[\cdot]\})\} \)
\( \cup[\cdot] \rightarrow W[\cdot] \)
\( \text{init}[\cdot] \rightarrow k \)
\( l[\cdot] = \text{max}(l[\cdot]) \)
\( S > S \cup \{u[\cdot], (\text{init}[\cdot])\} \)
\( \{\} \rightarrow \{\} \)
\( \{\} \rightarrow \{\} \)
\( \{\} \rightarrow \{\} \)

\(*\) Normalized Infixmap to be optimised: shaded area
Optimized and documented FlowMap

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)
c) Normalized Infomap: shaded areas to be filled in
### Conditional Statement

- **PreCondition**
  - $e \rightarrow e$
  - degree$[x] > b-1$
  - $x \leftarrow i$
  - degree$[i] > b-1$
  - degree$[i] = 1$
  - $i = x$

- **Impertive Statement**
  - $D := \Pi^* T$
  - $[T_{[e]} := 0$
  - $D := [x,y]$
  - $[T_{[w]} := wxy$

### Transition

- $e$ not an element of $S$
- $e$ removed from $S$
- edge not equal edge $ij$
- $x$ & $y$ elements of $T_i$ and $T_j$ resp.
- degree of $x$ gt $b-1$
- degree of $y$ lt $b-1$
- $y$ not equal node $i$
- degree of $j$ equals $1$
- $x$ not an element of $E$
- node $i$ equals node $x$

### Action

- Do not consider $w$, any more
- let $T_i$ & $T_j$ be subtrees
- update product matrix, $D$
- update final weights

### PostCondition

- normalized Infomap: state diagram (unshaded area)
f) Normalized infoMap to be optimised: shaded area
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\_dcsit() 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{align*}
& \text{begin} \\
& \text{dequeue}(Q_1, e_{ij}). \\
& \text{if degree}[i] = 1 \text{ or degree}[j] = 1 \text{ then} \\
& \qquad \text{enqueue}(Q_1, e_{ij}), \text{ let } t_i = t_i/(1 + \beta t_i), \text{ goto (5)}. \\
& \text{end} \\
& \text{loop } |E - S| \text{ times} \begin{align*}
& \text{begin} \\
& \text{dequeue}(Q_2, e_{xy}). \\
& \text{if degree}[x] = b \text{ or degree}[y] = b \text{ then enqueue}(Q_2, e_{xy}), \text{ goto (17)}. \\
& \text{if } e_{xy} \text{ cannot connect the two subtrees created by removing} \\
& \quad e_{ij} \text{ from } S, \text{ then enqueue}(Q_2, e_{xy}), \text{ goto (17)}. \\
& \text{let } \Delta = w_{xy} - w_{ij}, \text{ r be a random number in } [0,1]. \\
& \text{if } \Delta \leq 0 \text{ or } e^{-\Delta/\beta} > r \text{ then} \\
& \text{begin} \\
& \quad \text{let } S = S - \{e_{ij}\} \cup \{e_{xy}\}. \\
& \quad \text{enqueue}(Q_1, e_{xy}), \text{ enqueue}(Q_2, e_{ij}). \\
& \text{let } t_i = t_i/(1 + \beta t_i). \\
& \text{end} \\
& \text{enqueue}(Q_2, e_{xy}), \text{ let } t_i = t_i/(1 + \beta t_i). \\
& \text{end } \{\text{loop}\} \\
& \text{end} \\
& \text{Return } S \text{ and its cost for the best solution encountered.}
\end{align*}
Fig. 5.2.3.6 Data flow for Anneal algorithm
a) source code: original (see page 158)

1: Use gen_dcsf() 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}
11: Return $S$ and its cost for the best solution encountered.

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

#### Input
- \( S \) (a set of edges)
- \( E \) (a set of edges)

#### Output
- \( S' \) (a set of edges)

#### Statement
1. Let \( S' \) be an empty set.
2. While there is an edge in \( S \)
   - Select an edge \( (x, y) \) from \( S \)
   - Add \( (x, y) \) to \( S' \)
   - Remove \( (x, y) \) from \( S \)
3. Return \( S' \)

#### Action
1. Let \( Q1 \) be empty.
2. Let \( Q2 \) be empty.
3. Let initial temperature be 0.
4. Repeat for \( i \) from 1 to \( T \)
   - Choose temperature \( T \)
   - Let new weight \( w' \) be a random number in \([0, 1]\)
   - Update set of edges in \( DCST \)
   - Return set of edges in \( DCST \)
5. Return set of edges in \( DCST \)
a) Normalised Ink tag: state diagram (unshaded area)
0. Initialized queues
   edges to queue A
   no more edges for A
   new t = final t
   new t > final t
   edge e to queue A
   degree of node i equals 1
   degree of node j equals 1
   both degree of i and j not equal 1
   more edges to remove from Q2
   degree of nodes k and l equals b
   degree of nodes m and n equals a
   both degree of k and l not equal b
   enqueue edge to Q2
   calculate wt. differences
   update set of edges in DCST
   update set of edges in DCST
   S remains unchanged

11. update set of edges in DCST
    return S & its cost
    1: Start generation of S
    2: enqueuing edges to Queue 1
    3: enqueuing edges to Queue 2
    4: evaluating cooling schedule
    5: dequeuing edges from Queue 1
    6: computing initial cooling
    7: looping [E-S] times
    8: enqueuing edges to Queue 2
    9: calculating wt. differences
    10: updating set of edges in DCST
    11: returning S & its cost

[Conditional Statement]

[Procedural Statement]

[Imperative Statement]
Note: Steps - Optimized state diagram as Structured English (generated)
and source code (generated) - are to be done as in Fig. 5.2.4.1
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
    PATTERN[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 edgpoint.]
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. i.e. it is not a safepoint.]
                    if EDGEPOINT(n[j],n[j+4],border) then
                        [test each dark point to see if it is an edgpoint.]
                        begin
                            if SAFEPOINT(n,border) then
                                [Test each edgpoint 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 composition 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 PATTERN[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. i.e. it is not a safepoint.)
          if EDGEPOINT(n[j], n[j+4], border) then
            (test each dark point to see if it is an edgepoint)
            begin
            end
          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]
          end
          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
      end
    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;

b) edited source code: states 1 .. 6 identified
c) Normalized infoMap: shaded area to be filled in

O . . . . . . . . . . . . . . . . . . . x [Path]
S 0 0 0 0 0 0 0 0 0 0 10 [Transition]
    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 d     2:
d s s               3:
d s s               4:
d s s               5:
d               6:

row = first_row to last_row
column = 1 to MAXCOLUMN
DARK(p)
EDGEPOINTER(n[1],n[1],4,border)
SAFEPOINTER(n,border)

d = d + 1;
p = i * MAXINT
p = i


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

b) edited source code: states 1 .. 3 identified
c) Normalized infoMap: shaded area to be filled in

O . . . . x [Path]
S O O O O 4 [Transition]
  0 . . .
  . 0 . .
  . 0 . .
  . 0 . .
  . 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[i])
    d . 1 . . either a right or top edgepoint
  WHITE(n[i+4])
    . . 1 . 1 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)
e) Normalized infoMap: state diagram (unshaded area)

f) Normalized infoMap to be optimized: (shaded area)
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 SAFEPO\textsc{i}NT(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.\}
          \ (n[6] + not(n[7])) \ (n[2] + not(n[1]));
      2 : \{Evaluate for a top safepoint.\}
        SAFEPO\textsc{i}NT := 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.\}
        SAFEPO\textsc{i}NT := 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.\}
        SAFEPO\textsc{i}NT := not(n[2]) \ (n[3] + n[4] + n[0] + n[1])
          \ (n[4] + not(n[5])) \ (n[0] + not(n[7]));
      end; \{case\}
   end

\textbf{a) source code: original}

function SAFEPO\textsc{i}NT(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.\}
            \ (n[6] + not(n[7])) \ (n[2] + not(n[1]));
      2:
        2 : \{Evaluate for a top safepoint.\}
          SAFEPO\textsc{i}NT := 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.\}
          SAFEPO\textsc{i}NT := 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.\}
          SAFEPO\textsc{i}NT := not(n[2]) \ (n[3] + n[4] + n[0] + n[1])
            \ (n[4] + not(n[5])) \ (n[0] + not(n[7]));
      end; \{case\}
   end

\textbf{b) edited source code: states 1 .. 5 identified}
(Conditional Statement)
border = 0
border = 2
border = 4
border = 6

(Imperative Statement)
(n[6] + not(n[7])) (n[2] + not(n[1]));
(n[0] + not(n[1])); (n[4] + not(n[3]));
(n[2] + not(n[3])); (n[6] + not(n[5]));
(n[4] + not(n[5])); (n[0] + not(n[7]));

Action 1
right safepoint
1 top safepoint
1 left safepoint
1 bottom safepoint

<table>
<thead>
<tr>
<th>State</th>
<th>L L L L L L L L</th>
</tr>
</thead>
</table>
s s s 1: | E1 |
d s s 2: |
d s s 3: |
d d d d 4: |

(c) Normalized infoMap: shaded area to be filled in

(Conditional Statement)
border = 0
border = 2
border = 4
border = 6

(Imperative Statement)
(n[6] + not(n[7])) (n[2] + not(n[1]));
(n[0] + not(n[1])); (n[4] + not(n[3]));
(n[2] + not(n[3])); (n[6] + not(n[5]));
(n[4] + not(n[5])); (n[0] + not(n[7]));

Action 1
evaluate right safepoint
1 evaluate top safepoint
1 evaluate left safepoint
1 evaluate bottom safepoint

<table>
<thead>
<tr>
<th>State</th>
<th>L L L L L L L L</th>
</tr>
</thead>
</table>
s s s 1: | E1 |
d s s 2: |
d s s 3: |
d d d d 4: |

(d) Normalized infoMap: states and transitions added [shaded area]
c) Normalised infoMap: state diagram (unshaded area)

O . . . . . . . . . . . x [Path]
S 0 0 0 0 0 0 0 0 0 8 [Transition]
  o  right salepoint
  .  .  not a right salepoint
  .  .  top salepoint
  .  .  not a top salepoint
  .  .  left salepoint
  .  .  not a left salepoint
  .  .  bottom salepoint
  .  .  not a bottom salepoint
  .  .  .  5 [State]
  s s 1 start evaluation of right salepoint
  .  d s s 2 evaluation of top salepoint
  .  .  d s s 3 evaluation of left salepoint
  .  .  .  d s s 4 evaluation of bottom salepoint
  .  .  .  .  d d d d d d 5 exit
  .  G G G G G G G G 4 [preCondition]
  .  .  .  .  t f . . . . right salepoint
  .  .  .  .  t f . . . . top salepoint
  .  .  .  .  t f . . . . left salepoint
  .  .  .  .  t f . . . . bottom salepoint
  .  .  .  .  .  .  .  .  .  .  4 [Action]
  .  .  .  .  .  .  .  .  .  .  1 evaluate right salepoint
  .  .  .  .  .  .  .  .  .  .  evaluate top salepoint
  .  .  .  .  .  .  .  .  .  .  evaluate left salepoint
  .  .  .  .  .  .  .  .  .  .  evaluate bottom salepoint
  .  .  .  .  G G G G G G G G . [postCondition]

[Conditional Statement]
border = 0
border = 2
border = 4
border = 6

[Imperative Statement]
  (n[6] + not(n[7])) . (n[2] + not(n[1])),
  (n[0] + not(n[1])) . (n[4] + not(n[3])),
  (n[4] + not(n[5])) . (n[0] + not(n[7]));

f) Normalised infoMap to be optimized: shaded area
Conditional Statement

<table>
<thead>
<tr>
<th>Action</th>
<th>preCondition</th>
<th>postCondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

g) Optimised and documented InfoMap

Note: Steps - Optimised 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