A PASCAL Program to Convert Extended Entry Decision Tables Into Optimal Decision Trees

T. C. Layman

MLI 79-7

A publication of the *Machine Learning and Inference Laboratory*
Artificial Intelligence Center
George Mason University
Fairfax, Virginia 22030-4444 U.S.A.
(703) 764-6259

Editor: R. S. Michalski
Assistant Editors: J. Ralston and J. Zhang

The *MLI Reports* replace *ISG Reports*
published until December 1987
by the Artificial Intelligence Laboratory,
Department of Computer Science
University of Illinois,
Urbana, Illinois 61801

1979 - 7

PASCAL PROGRAM TO CONVERT EXTENDED ENTRY DECISION TABLES INTO OPTIMAL DECISION TREES

by Thomas Charles Layman

Internal Report

October 1979

1. Introduction

This report is concerned with the PASCAL implementation of two algorithms by Michalski [1] which convert unordered extended entry decision tables into space or time optimal decision trees based on the criteria of Minimizing Added Leaves (MAL) and Dynamically Minimizing Added Leaves (DMAL) formulated in [1]. The basic idea of these algorithms is: the rules of the table are transformed into elementary cartesian complexes, which are defined in Appendix A. The complexes are assigned to action classes which represent zero or more actions. The case of zero actions arises when you want to insure that no action is performed for those events. If an action class has more than one action, the actions should be independent of one another. In the case of the MAL criterion, the algorithm starts by finding an optimal cover of the complexes for the purpose of merging complexes which might be expressed as a single complex.

Next, a test is chosen to split the event space of the complexes according to either the MAL or the DMAL criterion. Briefly stated, these are:

a) MAL

1) Choose the test which has the smallest DELTA0, defined as the number of complexes broken by the test. A complex is not counted as broken if splitting the complex does not separate events of the original table from which the complex was formed.

2) In case of a tie, choose the test with the most outcomes.

3) If we still have a tie, choose the test which partitions the event space into parts with smaller complexes.

4) Choose any test if a tie still exists.

b) DMAL

1) This criterion only looks at action classes whose events are separated by the test being examined. For the action classes considered, determine the cardinality, c(E), of the cover of the complexes in these classes. Next determine the optimal covers within the smaller event spaces created after splitting on the test in question. Then subtract c(E) from the cardinality of the covers of the considered action classes in the smaller event spaces. The resulting value is called DELTA1. Choose the test with the smallest DELTA1.

2) Same as MAL 2).

3) If a tie still exists, choose the test which partitions the event space into parts in which covers of the same action class are larger.

4) Same as MAL 4).

There is a DMAL "shortcut" which states that if we determine a test to have a DELTA0 of zero, then DELTA1 must also be zero and we can skip the cover generation.

After a test is chosen, it is removed from the set of available tests. This test now represents an inner node in the optimal tree. The event space is split into smaller spaces by the test. These event spaces are further split by invoking the algorithm recursively until a space with one action class (i.e. a leaf) is reached. Note that the DMAL criterion forms optimal covers at each step of the process. I have appended to the MAL criterion an additional tie-breaking decision which produces trees in certain instances as good as DMAL without the large expense of generating optimal covers every time a splitting variable must be chosen. This will be described later.

As an illustration, suppose we were to apply MAL and DMAL to the diagram in Fig. 1. First, consider MAL. Test X1 would break 1 complex. X2 would break all 15 complexes. Tests X3 and X4 each break 2 complexes. So X1 would be picked. If we used DMAL, we would break one complex by X1 into 3 complexes. But on forming a cover in the X1=0 region, 2 complexes would combine, thus giving a DELTA1 value of 1. X2 would have a DELTA1 of 30. X3 and X4 each would yield a DELTA1 of 4. So again, X1 would be chosen.

The program accepts input tables in two modes: a) interactive mode which allows a user to enter the parameters and rule values at a terminal, and b) file mode which uses a prepared, formatted file containing the same information the interactive user inputs. In file mode, the data can be set up in advance and can be saved for repeated use if modifications to the data are desired. The resulting tree is output as a graphic tree or as PASCAL code using -case- "switches" for the internal tree nodes and procedure calls as the actions within the action class. The procedures the actions represent should be independent of the order in which they are called. This is a "stand alone" program, but if the user wishes, the tree-constructing procedures and important global variables may be removed for other uses. Fig. 2 illustrates interactive mode for a rather trivial example which would fit on one page.

The compiled program occupies about 8K 60-bit Cyber 175 words of object code.

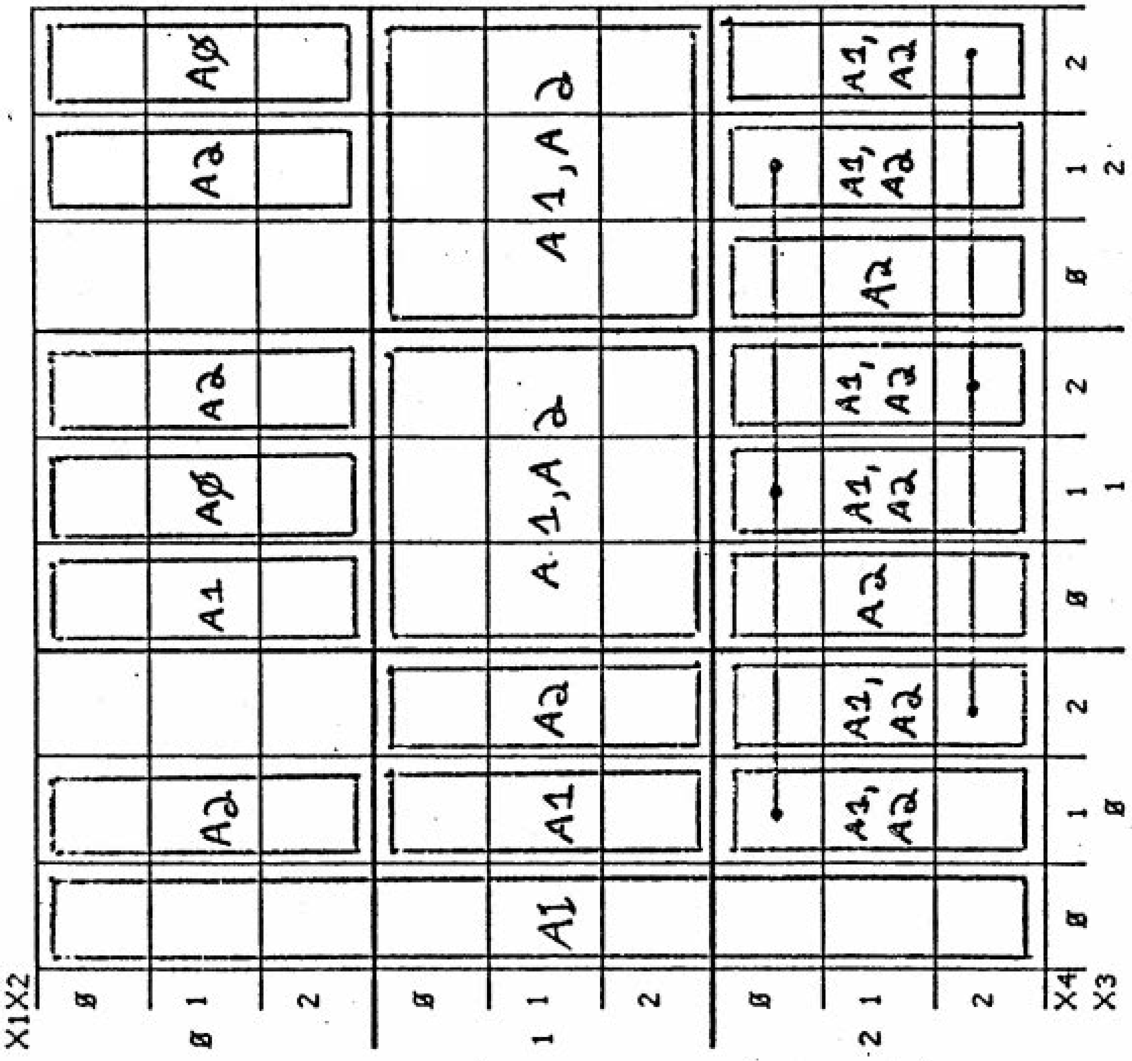

Fig. 1

```
INTERACTIVE OR FILE MODE?
TYPE I OR F
? i
 HOW MANY TESTS?
? 1
 WHAT MAX VALUE FOR TEST  1? (FROM 1 TO 14)
? 2
 HOW MANY ACTIONS?
? 1
 ENTER VALUES OF TESTS
 FROM THE RANGE LISTED OR -1 FOR DONTCARES
 FOR THE ACTION VALUES, ENTER 1 IF THE ACTION WILL BE TAKEN
 OR 0 OTHERWISE
 -- ENTER RULE   1 --
 TEST  1 (0 TO  2)
? 1
 NOW ENTER THE ACTION VALUES
 ACTION  1
? 1
 MORE RULES TO ENTER? TYPE Y OR N
? n
DIAGRAM DOESN'T COVER WHOLE EVENT SPACE.
DO YOU WISH TO ATTACH AN ACTION CLASS TO 'ELSE TREE NODE'?
TYPE Y OR N.
? n
 WHICH MODE? (T)IME OPTIMAL OR (S)PACE OPTIMAL?
? s
TIME TO FIND COVER(S):                2 MS

  -- FINAL COVER(S) --

 -- NEW ACTION CLASS --
 THIS CLASS CONSISTS OF
 ACTIONS [    1 ]
 THE COMPLEXES FOR THIS ACTION CLASS ARE
[X1=  1]

SIGMAO =  0


TIME TO GEN TREES =              5 MS
TOTAL PROC TIME:             7 MS

 TYPE D TO ENTER NEW DIAGRAM WITH SAME PARAMS
      P TO CHANGE PARAMS
      O TO PRINT OUT PARAMS
      Q TO QUIT
?
X1 -- 0 -- A[]
!
!
!---- 1 -- A[1]
!
!
 ---- 2 -- A[]
?
```

Fig. 2

2. Cover generation

While the recursive test determination dominates these algorithms, it is actually the optimal cover generation which consumes the most processor time. Even in the MAL method, where only one optimal cover is constructed, finding the cover may take several times the processing as does the tree generation. Part of the problem stems from the fact that covering the cells in different orders may produce different covers, especially if there are "don't care" events which may be consumed in different ways by the covering complexes. Because of the "combinatorial explosion", we obviously can't consider covering the individual cells in all possible orders. A natural alternative to this is to produce covers for all permutations of the action classes. The number of permutations is the-number-of-action-classes factorial. So when we exceed three action classes, we have the same combinatorial problem. Generating 24 covers of 4 action classes is an expensive proposition, and there may not be enough don't cares to warrant doing so many covers. For this reason, the user has the option of skipping the cover generation for a certain number of loops through the permutation algorithm when the input diagram is not complete (i.e. it doesn't cover the whole event space) and the number of action classes exceeds 3. There is the risk of a less optimal cover being produced, but many seconds of processing time can be saved. This "skip factor" always may be set by the user. But when there are more than 3 action classes, the user is alerted and given one more chance to set it.

In order to facilitate making good covers, the input complexes within an action class are sorted according to size, then broken into individual cells before invoking the optimal cover procedure. The ordering is necessary because one or more complexes adjacent to a larger complex (in the same action class) might "steal" a slice of cells from the larger complex to form a "don't care" selector which has the entire domain as its value, resulting in the decomposition of the large complex into a number of smaller ones (see Fig. 3). The complexes are broken into single cells because complexes which might be "orthogonal" may not be able to combine in their entirety, but may form better complexes after decomposition (see Fig. 4).

The covers are produced using the AQ procedure which forms the cover of a set E1 against a set E2 [2]. This is the same AQ procedure used by the INDUCE-1 system [3] with some modifications. Only nominal scale variables are used. Only two functions are used by procedure TRIM to find the "best" complex from the "intermediate star" covering an event. These functions are: the most "new covered" events in E1 and, in case of a tie, the most "don't cares". TRIM has been made external to AQ so that it can be used to sort complexes within an action class as described above. The function used for this sorting is only available when a flag, -aqmode-, is false. The -maxstaraq- parameter passed to AQ has been changed to be a variable rather than a constant. This is so in order to trim back the "intermediate stars" at the last possible moment. The low constant value from the INDUCE version often trimmed the star very early in the covering process when little information was known about the events in E1 and E2. Thus potentially good complexes were discarded. The value of -maxstaraq- is set to be the size of the sorting array

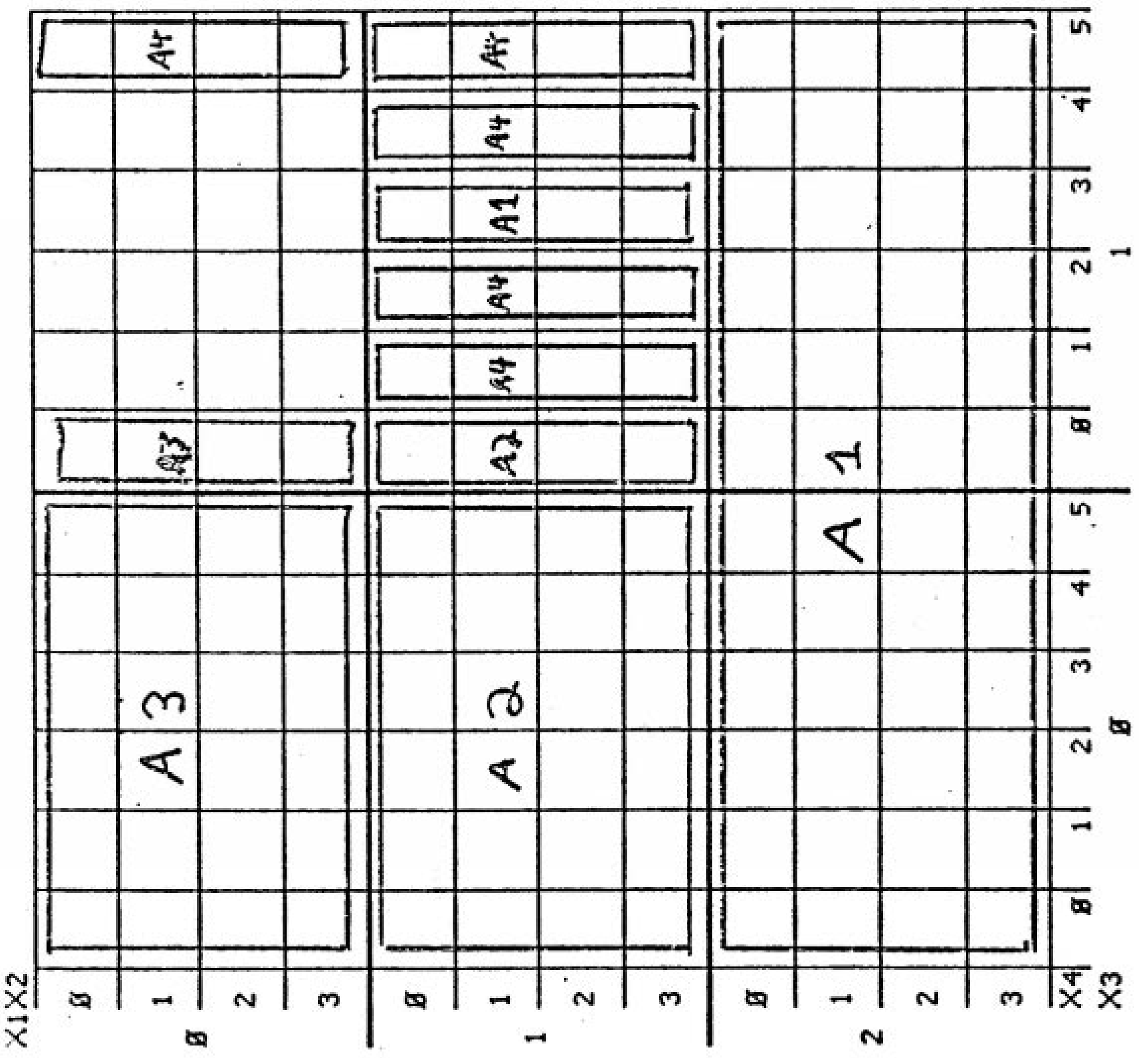

Fig. 3.a

Original complexes.

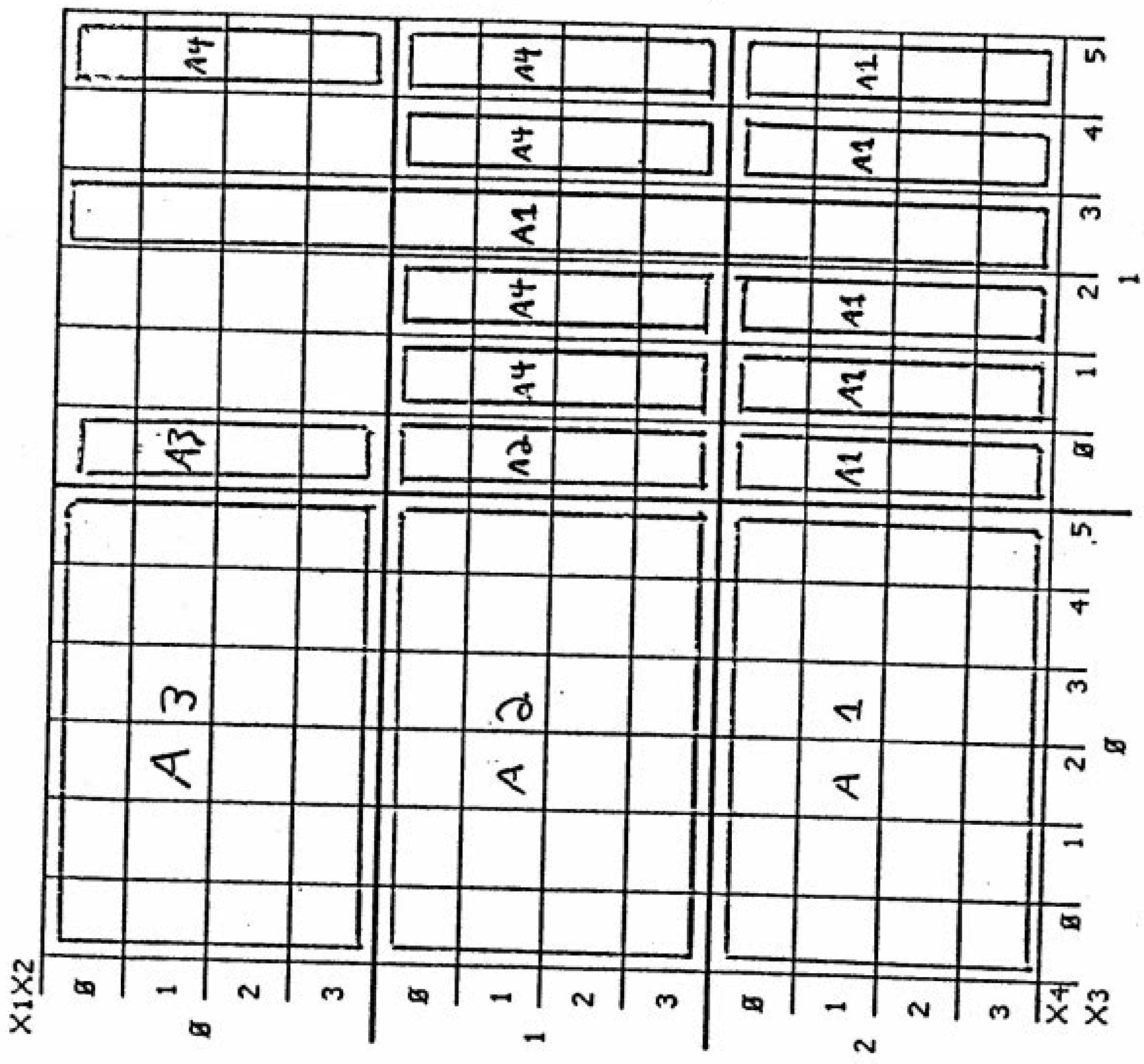

Fig. 3.b

After 'stealing'

a 'slice' of cells.

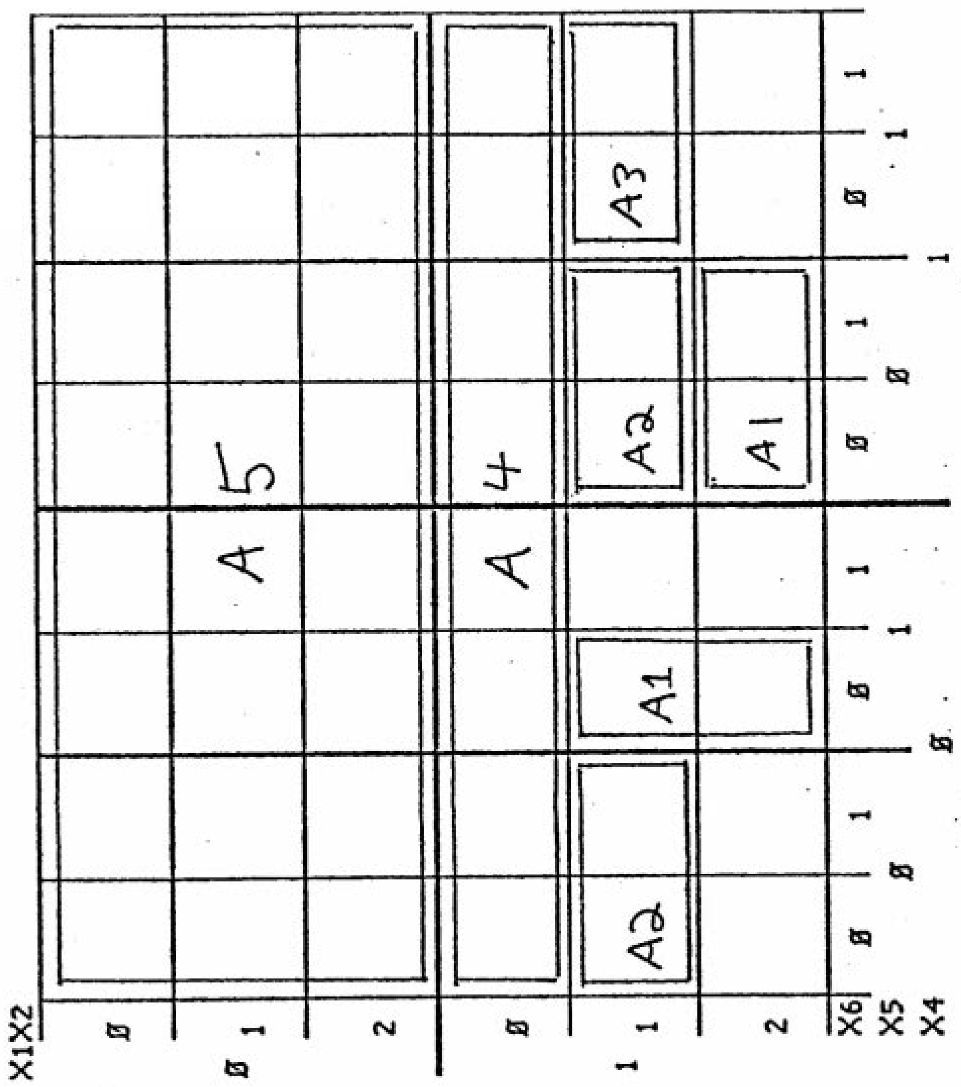

Fig. 4.a

Before breaking

the complexes.

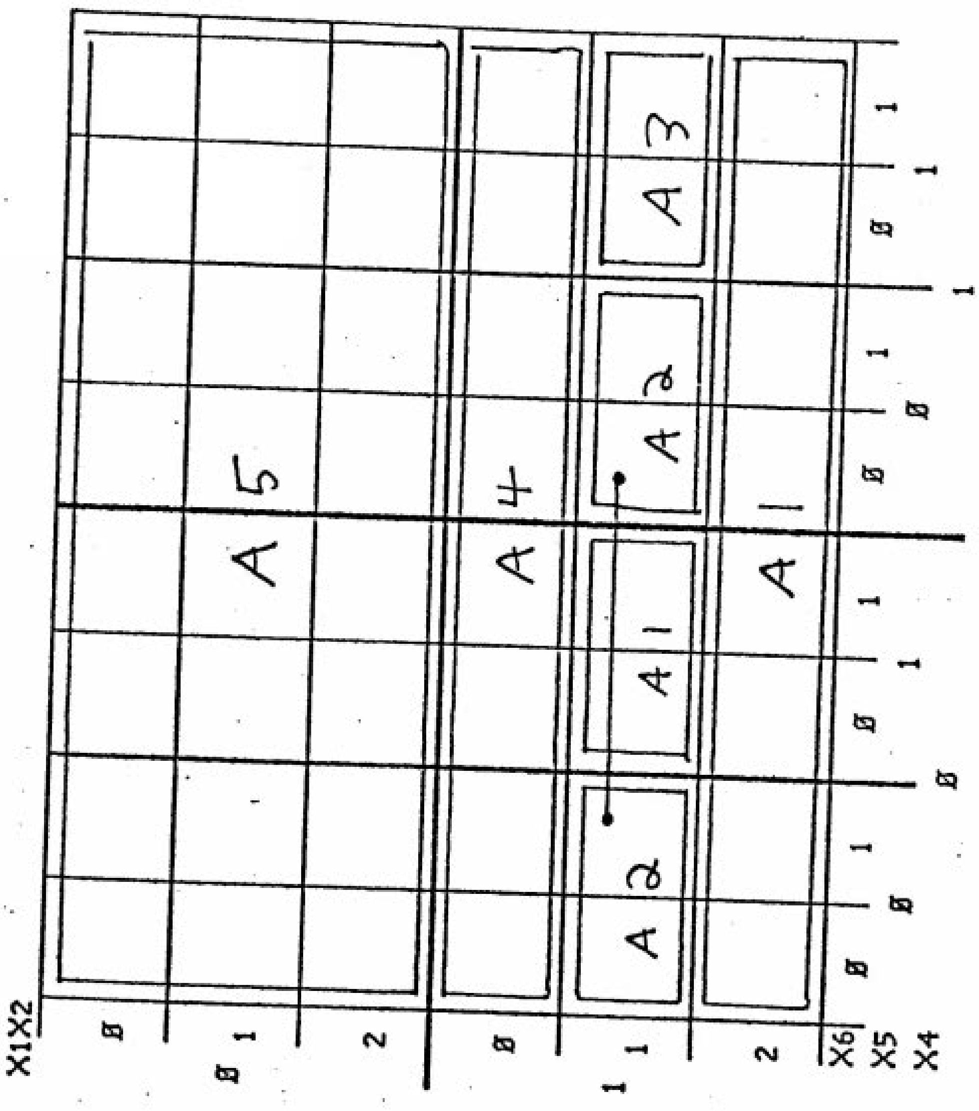

Fig. 4.b

After breaking up

the complexes.

divided by the number of input table variables (-nvar-), since the number of complexes in the star can increase, at most, by a factor of -nvar-. The last modification involves the requirement that complexes covering an action class do not intersect with each other. This is accomplished by placing a copy of each complex produced by AQ in the E2 set before looking for the next event to be covered.

The cover generator, OPTCOVER, finds covers for action classes one at a time. The set to be covered is passed as parameter E1. All cells covered by previous iterations, as well as the cells of the action classes to be covered, are enqueued as set E2. Since AQ often produces complexes with internal disjunction (with selectors that have more than one value, but less than the entire variable's domain), these complexes must be broken into "elementary complexes" (complexes with no internal disjunction). This is necessary for the case when there may be a complex of another action class occupying cells adjacent with respect to the selector with disjunction. When this happens, as is often the case, the MAL method MUST break the complex by the test which has disjunction in order to separate it from the complex of the other action class. This artificially inflates the value of DELTA0 (the number of broken complexes) for that test since the split will always be done. This may lead to choosing the improper test. Fig. 5 illustrates this. We MUST split on X3 in order to separate the complex of action class A1, [X1=2][X3=0,1], from the complex of action class 3.

If the user wishes, the cover generation can be skipped entirely and the tree formed with respect to the original decision table. The user may already have an optimal cover and re-generating it is unnecessary. Also, for a very large table, the cost of finding

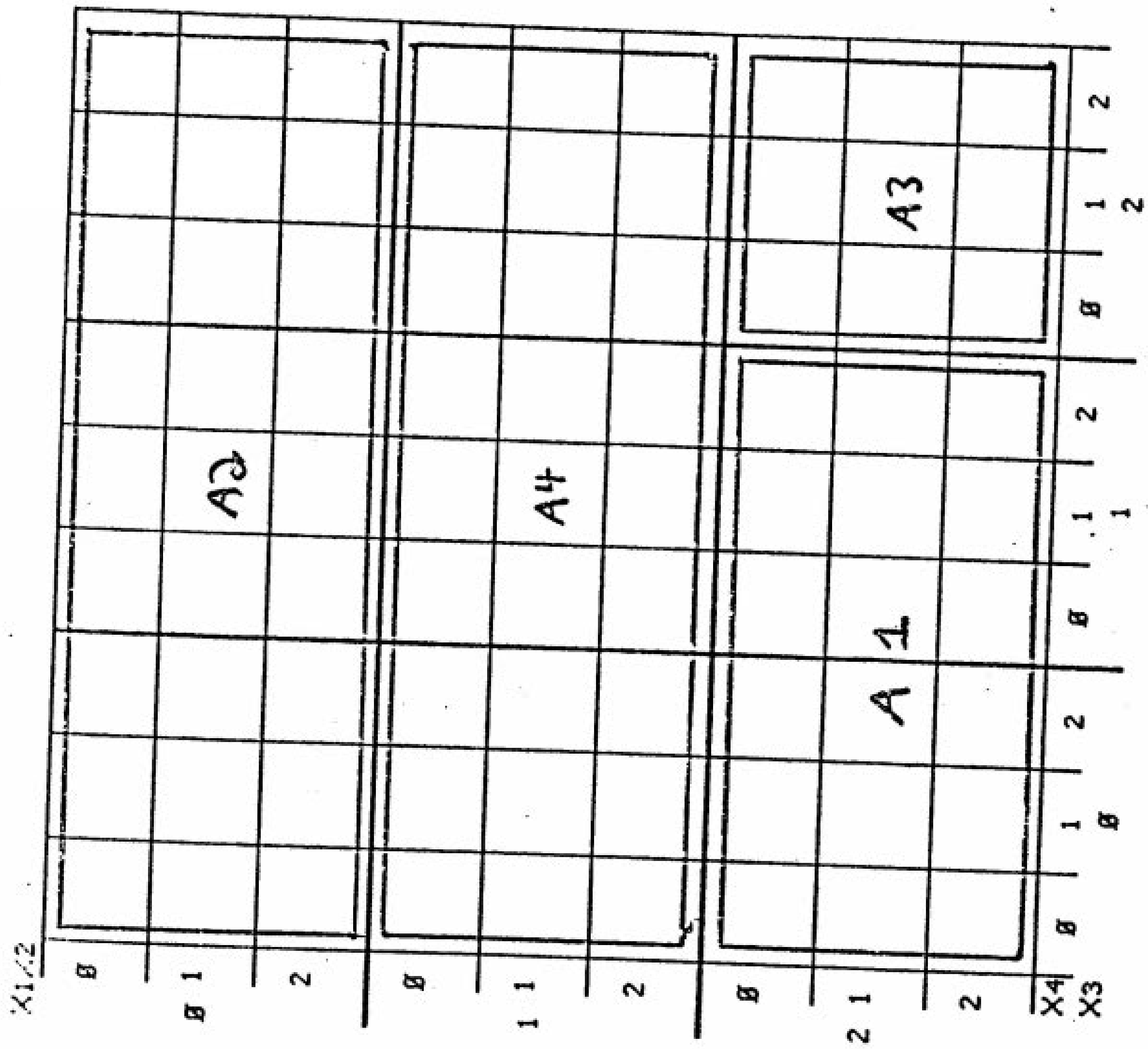

Fig. 5

an optimal cover may be prohibitive. In a 50-rule, 10-variable test example, the program ran about 1 minute before the Cyber aborted the job. Examination of the post-mortem dump showed that the cover generator was only on the first of six iterations!

3. MAL Modification

While the MAL criterion makes very good choices of test variables, it can make seemingly curious decisions in certain circumstances. Fig. 6 illustrates an example of a subdiagram that has been reached in the course of the tree algorithm. The complexes belong to action classes 1, 2 and 3, and are generalizations of the input table complexes denoted by the dotted circles. The choice looks clear. However, the MAL criterion doesn't count complexes as broken if cells from the original diagram aren't separated. In the example, tests 2 and 4 both have DELTA0 equal to 0. Both are tied in all the other tie-breaking criteria. Subdiagrams that contain no original diagram cells count only as "else leaves". So if test X2 is chosen arbitrarily, a subtree of 5 leaves will be produced rather than a 3 leaf tree when test X4 is chosen.

For this reason, a new tie-breaking rule has been appended to MAL to determine if any of the tied tests produces more "immediate leaves". We know that we have a leaf if the subdiagram contains only cells from a single action class. In this PASCAL implementation, this can be determined very quickly. Suppose we are checking the i-th variable. The "logical or" is taken of the i-th variable (encoded as a bitmask) for all events from the original table that are also in the subdiagram under consideration. This resulting mask is compared with two other bitmasks: -avail-, which represents

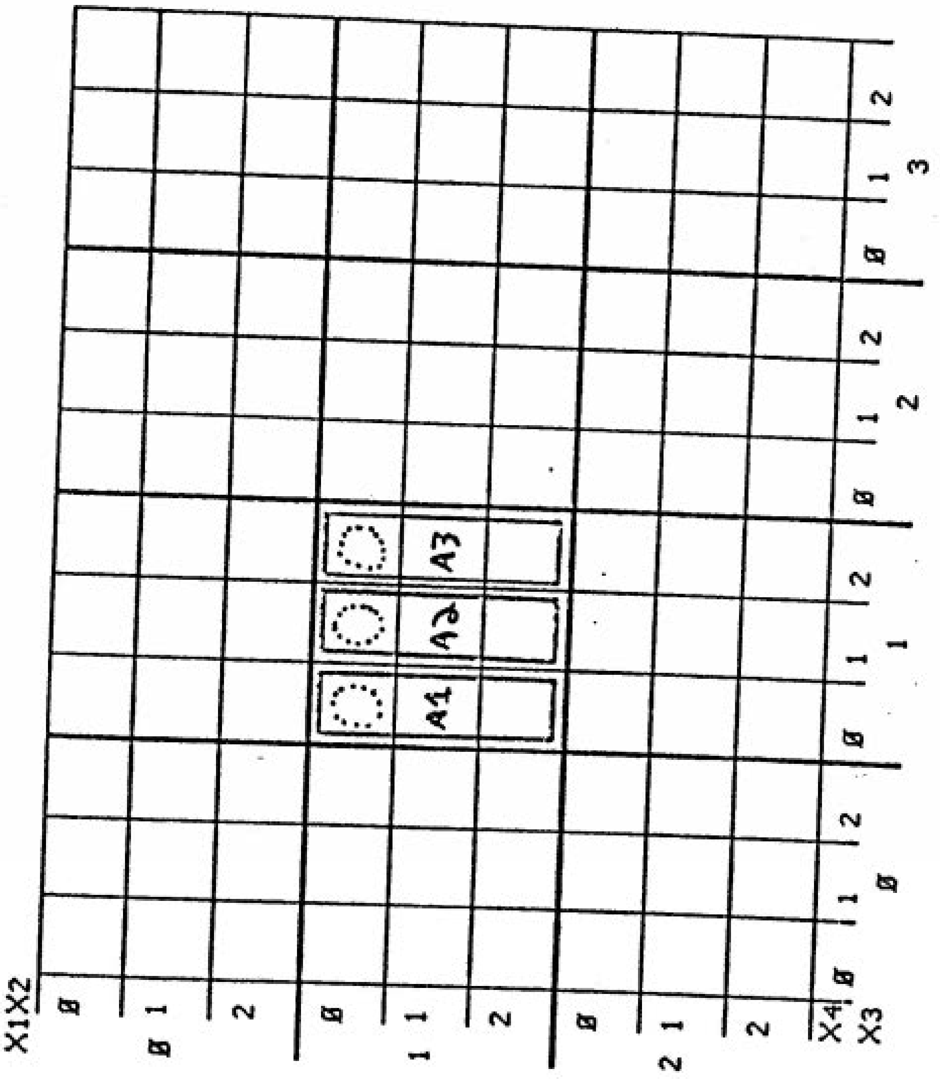

Fig. 6

the subdiagrams in which we have yet to find cells from the original table, and -onemask-, representing the subdiagrams where cells of only one action class have been found. For each action class examined, any bits set in both the test mask resulting from the "logical or" and -onemask- are removed from -onemask-. Any bits set in both the test mask and -avail- are removed from -avail- and added to -onemask-. When we are done, the cardinality of -onemask- gives the number of "immediate leaves".

This method will also find parts of broken complexes that can be combined, but only if the result is a leaf. Fig. 1 illustrates this feature. After splitting on X1, there are two complexes of action class 1 in the X1=0 subdiagram which can be combined to form a leaf, giving a subtree of 7 leaves in all. Without the modification, test X3 might be chosen, resulting in a 9-leaf subtree. The program's output for this example, in both tree and PASCAL form, is given in Fig. 7 and Fig. 8. If the complexes which can be merged are buried among other complexes so that a leaf can't be formed at that moment, the modified MAL test will not find them, although the DMAL method will detect them. Note also that the DMAL method would not have made the "mistake" that led to the added criterion.

4. Major Data Structures

The following are the main data structures of the program. The terms "complex", "action class" and "diagram" in the procedure descriptions which follow will refer to these structures and not the concepts which they represent.

A complex represents a Cartesian complex. Besides having an

```
X1 -- 0 -- X4 -- 0 -- A[1]
:          :
:          :---- 1 -- X3 -- 0 -- A[2]
:          :          :
:          :          :---- 1 -- A[]
:          :          :
:          :           ---- 2 -- A[2]
:          :
:           ---- 2 -- X3 -- 0 -- A[]
:                     :

:                     :---- 1 -- A[2]
:                     :
:                      ---- 2 -- A[]
:
:---- 1 -- X3 -- 0 -- X4 -- 0 -- A[1]
:          :          :
:          :          :---- 1 -- A[1]
:          :          :
:          :           ---- 2 -- A[2]
:          :
:          :---- 1 -- A[1,2]
:          :
:           ---- 2 -- A[1,2]
:
 ---- 2 -- X4 -- 0 -- X3 -- 0 -- A[1]
           :          :
           :          :---- 1 -- A[2]
           :          :
           :           ---- 2 -- A[2]
           :
           :---- 1 -- A[1,2]
           :
            ---- 2 -- A[1,2]
```

Fig. 7

```
CASE X1 OF
  0:   CASE X4 OF
       0:  A1;
       1:   CASE X3 OF
            0:  A2;
            1:  ;
            2:  A2;
            END;    (*  CASE X3   *)
       2:   CASE X3 OF
            0:  ;
            1:  A2;
            2:  ;
            END;    (*  CASE X3   *)
       END:    (*  CASE X4   *)
  1:   CASE X3 OF
       0:   CASE X4 OF
            0:  A1;
            1:  A1;
            2:  A2;
            END;    (*  CASE X4   *)
       1:   BEGIN
            A1;
            A2;
            END;
       2:   BEGIN
            A1;
            A2;


            END;
       END;    (*  CASE X3   *)
  2:   CASE X4 OF
       0:   CASE X3 OF
            0:  A1;
            1:  A2;
            2:  A2;
            END;    (*  CASE X3   *)
       1:   BEGIN
            A1;
            A2;
            END;
       2:   BEGIN
            A1;
            A2;
            END;
       END;    (*  CASE X4   *)
  END;    (*  CASE X1   *)
```

Fig. 8

array of nominal scale selector variables, it contains a pointer to a "next complex" for chaining.

An action class structure contains a bitmask which denotes the actions in the conceptual action class. There is a 0-th bit in the mask used as a marking bit. A pointer to the first complex in a chain of complexes for this class is included. There is also a pointer to the next action class.

A structure called a diagram consists of chained action class nodes with their respective complexes. The ordinary way of accessing a diagram is by a pointer to the first action class node. For the purpose of chaining together diagrams, there is a separate diagram header node which has a pointer to a diagram's first action class and a pointer to the next diagram header.

A tree node has a variant record which states whether the node is an internal node or a leaf. If it is a leaf, its appropriate action class is stored in a bitmask. If it's an internal node, an array of pointers points to its tree node "children" and the integer value of the test it represents is stored.

5. Major Procedures

The following conventions are used here: procedures are in capital letters. Variables are in lower case. Within the text, variable names are bracketed with hyphens. The following abbreviations are used: int = integer, bool = boolean, cptr = pointer to complex, aptr = pointer to action class node, dptr = pointer to diagram header, tptr = point to a tree node, bit = bitmask.

CSIZE(c: cptr)

is a function which returns the integer size of complex -c-.

TRIM(nstar: int; maxs: int; aqmode: bool; el: cptr)

sorts a chain of complexes pointed to by -nstar-. If -aqmode- is true, then the chain is sorted on two cost functions: a) number of cells newly covered by the complex, b) number of "don't cares". If -aqmode- is false, then we are sorting on complex size prior to decomposition for the covering routine. -El- is the event we are covering when TRIM is called from AQ.

AQ(fl, f2: cptr; maxstaraq: int)

is a function which returns a pointer to a chain of complexes which is the cover of a set of complexes pointed to by -fl- against a set of complexes pointed to by -f2-. Differences between this version and the INDUCE-1 version have been explained earlier in this report. AQ calls TRIM with -aqmode- set to true. The limit to the size of the "intermediate star" before we invoke TRIM is -maxstaraq-.

EXPAND(r, firstcl, lastcl: cptr; keepstar: bool)

creates a chain of complexes which exactly covers the set of events covered by -r-. If -keepstar- is true, the "expansion" of -r- is to a chain of elementary complexes. Internal disjunctions are "removed" in this way but "don't care" ("star") selectors remain intact. If -keepstar- is false, the expansion is to a chain of single cell complexes. -Firstcl- points to the head of the chain. -Lastcl- points to the last complex in the chain. The "seed" complex, -r-, remains unchanged.

DECOMP(a: aptr; subnac: int)

is a function which returns a pointer to the first action class node in a diagram. The resultant diagram is a decomposition into single cell complexes of the diagram whose first action class node is pointed to by -a-. The decomposition is done by calls to EXPAND with -keepstar- set to false. The variable parameter, -subnac-, is set to the number of action classes in the resultant diagram.

GETPARAM

is where the user sets the number of tests (-nvar-), the upper bounds of the various test domains (-nval(i)-) and the number of actions for the input table.

DTABLEIN

is a function which returns a pointer to the diagram representing the input table. It is here where the user inputs the rule values. The returned diagram is broken into single cells, but only if a cover is to be generated. As a side effect, the original unbroken diagram is pointed to be -origdiag-. This procedure calls DECOMP. It removes redundant cells if input rules from the same action class overlap. Inconsistancies are detected and aborted. Also the number of action classes is computed.

OPTCOVER(diagram: dptr; noties: bool; nac: int; tests: bit)

returns a pointer to a chain of diagrams which are optimal covers of the diagram pointed to by -diagram-. If more than one cover is found, only unique alternate covers are kept. If -noties- is false, only one cover is kept. The covers are generated by calls to AQ.

MAKESUBDG(subdiag, cover: aptr; i, t: int)

returns a diagram pointed to by -subdiag- which represents a subdiagram of -cover- split on the variable -t- for value -i- in the domain of -t-. Only one subdiagram is created.

PICKM(tests: bit; cover, origdg: aptr)

is a function returning an integer which is the best test for splitting the diagram pointed to by -cover- via the MAL criterion. The bitmask, -tests-, passes the tests available for selection. -Origdg- points to the original diagram so that we may determine if a test separates original cells within a complex. Global variables, -delta0- and -sigma0-, are set. These are the number of broken complexes and the summation of -delta0- over all tests chosen, respectively.

MAL(node: tptr; tests: bit; cover, origdg: aptr)

calls PICKM to determine the test by which we'll split -cover-. MAKESUBDG is called to form subdiagrams of the original diagram. If the subdiagram has only one action class or is null, the corresponding "child" of -node- is set to be a leaf node (an "else leaf" in the latter case). Otherwise, the chosen test is removed from -tests-, the available test set, and MAL calls itself recursively with the -node- parameter passed as tree nodes which are "offspring" of -node- in the calling procedure.

PICKD(cover, subcovs: aptr)

is a function internal to procedure DMAL which chooses from the set of available tests, which test best splits the diagram pointed to by -cover- via the DMAL criterion. The integer value of this test is returned. The variable parameter -subcovs- is also set to point to the chain of covers of subdiagrams which are formed as part of the DMAL selection

Note that Case 1 is Example 1 in [1] which is from a complete table (i.e. it covers the whole event space). The other cases were not from complete tables. DMAL was not used in cases 4 and 5 because of its prohibitive cost. In Case 5, because of its large size, no optimal cover was generated for MAL, but the tree algorithm was run on the "raw" decision table rules.

6. Future Considerations

After some further modifications, which will include test costs and action probabilities, a user manual for this program will be produced. Further research comparing these algorithms with other methods will be conducted.

Acknowledgements

I would like to thank my project advisor, Prof. Michalski, for his encouragement and useful suggestions.

This report was generated on the PLATO-IV system at the Computer-Based Education Research Lab (University of Illinois) using the Qume printer.

process. Procedures MAKESUBDG, DECOMP, and OPTCOVER are called. PICKD sets -delta1- and -sigma1- which is a summation of -delta1- over all chosen tests.

DMAL(node: tptr; tests: bit; cover, origdg: aptr)

calls PICKM to see if -delta0- equals 0 so the shortcut explained in [1] can be taken. If -delta0- is not 0, then PICKD is called. DMAL then calls itself using the chain of subdiagram cove which PICKD generates. The set of available tests is passed as -tests- and -origdg- points to the original diagram.

5. Program Performance

Case number.	1	2	3	4	5
No. of rules.	27	10	14	34	50
No. of tests.	6	4	6	7	10
No. of action classes.	6	4	3	3	3
Size of event space.	64	108	128	394	9456
MAL					
Time to find cover (ms).	70	2029	661	19998	--
Time to find optimal tree (ms).	45	51	57	269	479
Sigma0.	0	2	1	5	8
Number of leaves/ number of "else leaves".	6/0	17/2	9/0	36/4	79/18
DMAL					
Time to find optimal tree (ms).	--	3375	3398	--	--
Sigma1.	--	4	0	--	--
Number of leaves/ number of "else leaves".	--	17/2	9/0	--	--

Table 1.

Program performance data.

References

1. Michalski, R. S. Designing Extended Entry Decision Tables and Optimal Decision Trees Using Decision Diagrams. Report No. UIUCDCS-R-78-898, Department of Computer Science, University of Illinois, Urbana, IL., March, 1978.

2. Michalski, R. S. Synthesis of optimal and quasi-optimal variable-valued logic formulas. Proceedings of the 5th International Symposium on Multiple-Valued Logic, Bloomington, Indiana, May 13-16, 1975, 76-87.

3. Larson, James, INDUCE-1: an interactive inductive inference program in VL_{21} logic system, Report No. UIUCDCS-R-77-876, Department of Computer Science, University of Illinois, Urbana, May, 1977.

Appendix A

Definition of Cartesian Complex

A cartesian complex is defined over an event space $E = D_1 \times D_2 \times \ldots \times D_n$ where $D_i = \{0,1,2,\ldots,d_i\}$ for an integer d_i. A cartesian complex, C, is defined as:

$$C = \bigcap_{i \in I} \{x_i = \alpha_i\}$$

where $\{x_i = \alpha_i\}$, called a cartesian literal, is a set of all events $e = (x_1,x_2,\ldots,x_i,\ldots x_n) \in E$, such that the value of x_i is an element of α_i, $\alpha_i \subseteq D_i$.

An elementary complex is a cartesian complex where the α_i either take a single value from D_i or take the entire domain as their value. In the text, a cartesian complex is called, simply, a complex.