CONSTRAINED ENCODING IN HYPERCUBES: ALGORITHMS AND APPLICATIONS TO LOGICAL SYNTHESIS

by

Tiziano Villa

Memorandum No. UCB/ERL M87/37

21 May 1987

CONSTRAINED ENCODING IN HYPERCUBES: ALGORITHMS AND APPLICATIONS TO LOGICAL SYNTHESIS

by

Tiziano Villa

Memorandum No. UCB/ERL M87/37

21 May 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

PLA optimization aims at minimizing the area occupied by the PLA and the delay through it (proportional to the number of product-terms, to a first order approximation). PLA-based FSM optimum state assignment looks for the assignment corresponding to a PLA implementation of minimum area. State assignment by means of symbolic (multiple-valued) logic minimization requires solving the problem of CONSTRAINED CUBICAL EMBEDDING : assigning subsets of states to subcubes of a boolean k-cube, for a minimum k. Combinatorial theoretical models of CONSTRAINED CUBICAL EMBEDDING are examined. A review of previous techniques of embedding graphs into hypercubes is presented. A new model in terms of poset embedding is given and new optimization problems are defined : SUBPOSET DIMENSION and SUBPOSET EQUIVALENCE (the recognition version of the first one). The latter is shown to be NP-complete and the former is shown to be NP-hard.

An algorithm to solve exactly SUBPOSET DIMENSION is described . It has been developed, beside intrinsic combinatorial interest, as a tool to make real progress with approximate solutions to CONSTRAINED CUBICAL EMBEDDING based on heuristics. An approximate algorithm, NOVA, that tries heuristically to satisfy all it can with minimum code length (log of cardinality of the state set) is also described. Results of an extensive testing are reported, showing that NOVA outperforms KISS in terms of area efficiency and that it can handle also large examples on which KISS is unable to complete successfully. A table is reported with the results of the exact encoding algorithm applied to the optimal encoding of some FSMs . Compared to KISS, the exact algorithm achieves almost always a smaller number of product terms and a shorter code-length, but NOVA is almost always winner in terms of area .

Acknowledgements

My research advisor, Prof. Alberto Sangiovanni Vincentelli is greatly acknowledged for having introduced me to the study of logic synthesis. He taught me (and still is) how the mind of an applied mathematician works, showing me how to put together elegance of analysis with effectiveness of application. Through my years of graduate school at Berkeley he has always been an inexhaustible source of encouragement and advise.

Prof D.O. Pederson took the burden of reading this report, providing useful remarks; Prof. R. Newton contributed with many ideas to the logic synthesis group of Berkeley.

I wish to thank Luciano Lavagno (CSELT Labs, Torino, Italy) for collaboration in obtaining the experimental results and for lively arguments on many aspects of PLA-based FSM synthesis.

Richard Rudell provided examples and interesting discussions. Early discussions on optimal state assignment with Giovanni De Micheli are also acknowledged.

I am grateful to Fabio Romeo for (among other things) help in quickly enhancing my editorial skills to finish in time the report.

I acknowledge and greatly appreciate the support of CSELT Labs during the first year and an half of my graduate study. The heads of the IC-CAD group, Luciano Leproni and Amelio Patrucco, were always understanding of what was more proper to me. This research has been sponsored in part by (NSF) ECS-8430435 subcontract with the University of Colorado, MICRO, and Silicon Compilers, Inc.

TABLE OF CONTENTS

CHAPTER 1

Introduction

The design of digital circuits can be viewed as a sequence of transformations (depending on the design style being used) of design representations at different levels of abstraction . Parametrized modules or macrocells , because of their flexibility, are currently widely used to implement digital subsystems of Very Large Scale of Integration (VLSI) digital integrated circuits .

Parametrized macro-cells can implement functional units that are specified by design parameters and by their functionality . Macro-cells are highly regular and structured and so may be efficiently produced by module generators . Programmable Logic Array (PLA) macros provide a simple and regular layout strategy for combinational logic functions expressed in two-level canonical form and implemented on a two dimensional array . Sequential logic functions, usually modeled by Finite State Machines (FSM), can be implemented by a PLA for the combinational part and by latches for the feedback .

The automatic synthesis of a sequential circuit as a PLA-based FSM involves functional design, logic design, topological design and physical design . The step of functional design defines the system behaviour by means of a functional description given by a state table or equivalent formalisms . The step of logic design maps the functional description into a logic representation in terms of logic variables . A representation of the symbolic states (and also of the proper inputs and outputs, if they are symbolic) in terms of boolean variables, called state assignment, is chosen . The complexity of the combinational component of the FSM depends heavily on the state assignment and selection of memory elements . PLA optimization aims at minim-

izing the area occupied by the PLA and the delay through it (proportional to the number of product-terms, to a first-order approximation) .

The PLA area is proportional to the product of the number of rows (product-terms) times the number of columns . The optimum state assignment (or encoding) problem looks for the assignment corresponding to a PLA implementation of minimum area . The (minimum) number of rows is the cardinality of the (minimum) cover of the FSM combinational component according to a given assignment . The number of bits used to represent the states (and the proper inputs and outputs, in case they are symbolic) is related to the number of PLA columns . Therefore the PLA area depends in a complex way on the state assignment .

Two related and simpler optimal state assignment problems may be defined :

1) find the assignment of minimum code-length among those that minimize the number of rows of the PLA :

2) find the assignment that minimizes the number of rows of the PLA among those of given code-length .

The optimum solution to the state assignment problem is achieved trading-off between the solutions to the previous two problems .

Special cases of the general state assignment problem of PLA-based FSMs are the optimal encoding of the inputs of a PLA and the optimal encoding of the outputs of a PLA [Rud86] . The optimal encoding of the inputs of a PLA, with n binary inputs, 1 (or more) symbolic inputs and m outputs looks for an encoding of the symbols into binary vectors minimizing the number of product terms needed to represent the function in disjunctive normal form . The optimal encoding of the outputs of a PLA, where n binary inputs are assigned to one or more sets of symbolic outputs, looks for an encoding as binary vectors of the symbols in the output sets in order to minimize the number of product terms needed to represent the function in disjunctive normal

form .

Recent advances in the state assignment problem [BHM84] have made a key connection with multiple-valued logic minimization : the states of a FSM are represented as the set of possible values for a single multiple-valued variable . Logic minimization is applied on a symbolic (code-independent) representation of the combinational component of the FSM . The effect of symbolic (multiple-valued) logic minimization is to group together the states that are mapped by some input into the same next-state and assert the same output . The problem arises (called CONSTRAINED CUBICAL EMBEDDING) of assigning each of these sets to subcubes of a boolean k-cube, for a minimum k . This would allow to group together the state codes in binary-valued logical implicants in the same way states are grouped together in the minimal symbolic (multiple-valued) cover , to obtain a binary-valued cover of the FSM combinational component having as many implicants as the minimal symbolic cover . An encoding such that each subcube contains all and only the codes of the states included in the corresponding subset of states satisfies the above requirements . In fact, each coded implicant represents all and only the state transitions related to the corresponding symbolic implicant . This idea works well also for the optimal encoding of the inputs of a PLA (purely combinational logic) .

Notice that the state (or symbolic input) encoding operation transforms a minimal symbolic cover into a non-necessarily minimal boolean cover . This is because in the symbolic cover the components of the next-state function have disjoint on-sets, while, in the coded boolean cover, some state codes have a non-zero entry in the same position and therefore the components of the next-state functions are not necessarily disjoint. Therefore the boolean cover of the FSM may require fewer product-terms . So the original formulation optimizes the choice of the present-state codes, neglecting the implications of the corresponding next-state encoding .

More recent efforts of taking into account the next-state encoding [DeM86], [Vil86b] gave birth to a more full-fledged symbolic minimization technique that looks for a minimal encoding-independent sum-of-products representation of a symbolic function. A key observation is that (the size of) representations of symbolic functions depend on the definitions of operations among symbols and that (the size of) representation of results of operations depend on a linear order on the domain and range of the variables of the symbolic functions . In a sum-of-product representation only the order in the range affects the efficiency of the representation and when such a linear order relation holds, symbolic function representations are equivalent to multiple-valued representations . Since no order relation holds a priori among the symbols of the description, we have the degree of freedom of introducing an order to obtain the most efficient representation of the symbolic functions . Once a linear order on the range has been defined, we are left with the problem of finding an encoding of the states that satisfies it , together with the constraint relations coming from the inputs . The previous outlined solution works in principle also for the optimal encoding of the outputs of a PLA .

This report deals with combinatorial theoretical models and algorithmic solutions of the problem of the constrained encoding of subsets of states (or symbols) in hypercubes, and with their application to the optimal state assignment of finite state machines and other problems of logical synthesis . We are leaving out from the combinatorial analysis the problem of satisfying also the order relations among the next-state symbols, because we are still working on how to find efficiently a linear order among them and how to reach eventually the unification of next-state ordering and proper output phase assignment . The results of this work will dictate which combinatorial problem we really want to solve .

Work done in 1983 at UCB and reported in [MSV83] paved the way for the breakthrough-idea of connecting optimal state assignment and multiple-valued logic minimization . A program, called SAP, was implemented, whose approach was based on the use of distance relations among the codes of the states . Distance requirements to reduce the combinational logic, were determined by foreseeing the effect of a heuristic minimization of the combinational logic related to a symbolic description of the FSM, and were represented by a graph . Adjacent code assignment was pictured as an embedding of an adjacency graph into a boolean hypercube . To ease the embedding task, don't care conditions were exploited in state codes . States were coded associating each vertex of the adjacency graph to a subset of vertices of the boolean hypercube, i.e. embedding the adjacency graph into a squashed hypercube, having appropriate faces squashed into vertices . Difficulties with bounding the code-length plagued this approach .

After the breakthrough already mentioned , a program (KISS) [DBS85], [DeM83] implementing the previous approach was developed at UCB . KISS increases the code length as much as needed by its heuristic encoding strategy, to satisfy the lower bound on the product-terms cardinality .

In the spring 1986, we developed at UCB a new program NOVA [Vil86c], [Vil86a], coded in C (4000 lines of code) to overcome some practical limitations of KISS, but more seriously because of persuasion gained from experiments, that increasing too much the length of the codes of the states of a FSM, it isn't likely to pay too much in terms of area . So we decided to implement an encoding algorithm, that did not increase the length of the code to satisfy more constraints . NOVA tries heuristically to satisfy all it can with minimum code length (log of cardinality of the state set) . And the experimental results have been quite good .

After that NOVA achieved satisfactory results in practice, we grew unsatisfied with the little of sound combinatorial analysis that we could claim for its basic combinatorial optimization problem (CONSTRAINED CUBICAL EMBEDDING) . We investigated with some depth the treatment of embedding problems in the literature and we formulated a model of our own .

We report in Chapter 2 about combinatorial theoretical models of the problem of CONSTRAINED CUBICAL EMBEDDING . A review of previous techniques of embedding graphs into hypercubes is presented . Reasons why previous schemes didn't capture the combinatorics of the problem we are dealing with and had poor performances when applied in the context of the optimal state assignment problem are offered . A new model in terms of poset embedding is given and new optimization problems are defined : SUBPOSET DIMENSION and SUBPOSET EQUIVALENCE (the recognition version of the first one) . The latter is shown to be NP-complete and the former is shown to be NP-hard . The poset embedding model, very useful for our applications, is also an interesting tool for the investigation of algorithmic properties of intersecting families of sets . The analysis leading to the formulation of the SUBPOSET problems is highly suggestive of algorithmic solutions .

In Chapter 3 we describe such an algorithm to solve exactly SUBPOSET DIMENSION . We detail its architecture, and we discuss the complex efficiency issues involved in its design . The algorithm is based on a double backtracking cycle and on a decreasing cardinality constraint assignment order . It tries to discover as soon as possible that a partial solution cannot be extended to all the domain, and it does it with one level of look-ahead with respect to the intersections of the constraints currently coded . The reason for developing an exact algorithm, beside intrinsic combinatorial interest, is that to make real progress with approximate solutions to CONSTRAINED CUBICAL EMBEDDING based on heuristics, we need a tool for their validation .

We implemented a prototype of the algorithm . A table is reported with the results of the exact encoding algorithm applied to the optimal encoding of some FSMs, compared against KISS and NOVA. Compared to KISS, the exact algorithm achieves almost always a smaller number of product terms and a shorter code-length . In terms of area, NOVA wins in all reported examples, except one where the exact algorithm gives the best .

In Chapter 4, we present the underlying embedding algorithm of NOVA and we compare it with KISS. We report in the appendix the results of an extensive testing, showing that NOVA outperforms KISS in terms of area efficiency and that it can handle also large examples on which KISS is unable to complete successfully .

Chapter 5 concludes with directions of future work and final remarks .

Comparisons of NOVA versus KISS on a collection of industrial and academic FSMs and the software documentation of the program NOVA are given in the appendix

CHAPTER 2

A mathematical model of the problem

According to the scenario already outlined in the introduction , in this report we are mainly attacking the combinatorial optimization problem CONSTRAINED CUBICAL EMBEDDING : given a collection of subsets of states or symbols (sometimes called constraint relations) we want to assign each of these subsets to subcubes (called faces) of a boolean space of minimum dimension in such a way that each face is a subspace which does not intersect the boolean vector (called encoding, code or assignment) assigned to any state not contained in the corresponding constraint . Formally it may be stated as :

INSTANCE : Set $S = \{1,....,n\}$ and a collection $C(S_i)$ of subsets $S_i \subseteq S$, $i = 1,....,m$.

QUESTION : find the minimum k and an injective map f from the sets $\tilde{S}_i \in C(S_i) \bigcup S$ to $\left\{0,1,x\right\}^k$

$$f : \tilde{S}_i \longrightarrow f(\tilde{S}_i)$$

such that for all subsets $S_i \in C(S_i)$, $s \in S$:

$$f(S_i) \bigcap f(\left\{s\right\}) \neq empty \text{ } iff \text{ } s \in S_i .$$

The purpose of this chapter is to analyze combinatorial theoretical models of the problem of CONSTRAINED CUBICAL EMBEDDING . A review of previous techniques of embedding graphs into hypercubes is made . We will see the reasons why previous schemes didn't capture the combinatorics of the problem we are dealing with and had poor performances when applied in the context of the optimal state assignment problem . A new model in terms of poset embedding is given and a new NP-hard

optimization problem is defined : SUBPOSET DIMENSION . The proposed model will lead later to the design of an algorithmic solution to our problem .

2.1. A guided tour of graph embedding problems

To appreciate the complexity of CONSTRAINED CUBICAL EMBEDDING and the intricacy of its "family tree" it is worth exploring briefly the "graph embedding" world . We will restrict our attention to models related to CONSTRAINED CUBICAL EMBEDDING and we will point out their computational complexity . We will concentrate on cubical graphs (subgraphs of hypercubes), the key combinatorial objects in which we would like to be able to embed our graphs. Later we will see their algorithmic treatment in applications of our interest and we will assess merits and drawbacks . This will prepare the ground for proposing eventually the formulation that captures better, at the moment, the combinatorics of CONSTRAINED CUBICAL EMBEDDING and lends itself to the design of efficient algorithmic solutions .

The graph embedding world includes the many combinatorial problems that ask questions about the embedding of graphs into other objects . Most of them are NP-complete . A problem is defined to be NP-complete if it belongs to NP and all other NP problems are polynomially transformable to it, or briefly transform to it . NP is class of all decision problems, that under reasonable encoding schemes, can be solved by polynomial time nondeterministic algorithms. To prove that a problem is NP-complete it is sufficient to show that it belongs to NP and that some known NP-complete problem transforms to it . Among the embedding problems of most interest for us, we recall :

SUBGRAPH ISOMORPHISM [GaJ79]

INSTANCE: Graphs $G=(V_1,E_1)$, $H=(V_2,E_2)$

QUESTION: Does G contain a subgraph isomorphic to H, i.e. a subset $V \subseteq V_1$ and a subset $E \subseteq E_1$, such that $|V| = |V_2|$, $|E| = |E_2|$ and there exists a one-to-one function $f : V_2 \rightarrow V$ satisfying $\{u,v\} \in E_2$ iff $\{f(u),f(v)\} \in E$?

Computational complexity: NP-complete in the general case ; CLIQUE transforms to it.

SUBGRAPH HOMEOMORPHISM [GaJ79]

INSTANCE: Graphs $G=(V_1,E_1)$, $H=(V_2,E_2)$

QUESTION: Does G contain a subgraph homeomorphic to H, i.e. a subgraph $G'=(V',E')$ that can be converted to a graph isomorphic to H by repeatedly removing any vertex of degree 2 and adding the edge joining its two neighbors ?

Computational complexity: NP-complete in the general case; SUBGRAPH ISOMORPHISM is a special case of it .

SUBGRAPH HOMOMORPHISM [Joh82]

INSTANCE: Graphs $G=(V_1,E_1)$, $H=(V_2,E_2)$, with self-loops allowed but no multiple edges .

QUESTION: Is there a homomorphism from G to a subgraph of H, i.e. a function $f : V_1 \rightarrow V_2$ such that if $\{u,v\} \in E_1$ then $\{f(u),f(v)\} \in E_2$?

Computational complexity: NP-complete; GRAPH 3-COLORABILITY transforms to it . This problem differs from graph isomorphism mainly in that the image of G need not be isomorphic to H but instead can be a proper subgraph (if H is loop-free the problems are otherwise equivalent) .

GRAPH ENCODABILITY [Joh82]

INSTANCE: Directed graphs $G=(V,A)$, $H=(V',A')$

QUESTION: Is there an encoding of G in H , i.e. a one-to-one function $f : V \rightarrow V'$ such that, for each arc $(u,v) \in A$, there exists a directed path from $f(u)$ to $f(v)$ in H .

Computational complexity: NP-complete; 3-SAT transforms to it . This problem differs from the directed SUBGRAPH HOMEOMORPHISM problem because the paths corresponding to edges need not be vertex-disjoint . This problem had its source in simulating a data structure with another [Ros78] .

UNIFORM GRAPH ENCODABILITY [Joh82]

INSTANCE: Directed graphs $G=(V,A)$ and $H=(V',A')$, sets L,L' of labels with $|L|=$ maximum out-degree of G and $|L'|=$maximum outdegree of H, labeling functions $h : A \rightarrow L$ and $h' : A' \rightarrow L'$ such that no two arcs with the same initial vertex get the same label.

QUESTION: Is there a uniform encoding of G in H, i.e. a pair of one-to-one maps $f : V \rightarrow V'$ and $e : A \rightarrow \{$paths in $H\}$ such that

(i) for all $a=(u,v)$ in A, $e(a)$ is a path from $f(u)$ to $f(v)$ and

(ii) if a_1 and a_2 are arcs in A with the same label, then the sequence of labels on the path $e(a_1)$ is the same as sequence on the path $e(a_2)$?

Computational complexity: NP-complete; FINITE STATE AUTOMATA INTERSECTION transforms to it . PSPACE-complete .

EMBEDDING DIMENSION [Joh82]

INSTANCE: Graph $G=(V,E)$, positive integer K.

QUESTION: Does G have embedding dimension K or less, i.e. is there a one-to-one function $f : V \rightarrow \{0,1,2\}^K$ such that, for all u,v in V , $\{u,v\} \in E$ iff $f(u)$ and

$f(v)$ differ by at most 1 in any component ?

Computational complexity: NP-complete; VERTEX COVER transforms to it .

2.2. Cubical graphs

Earlier techniques of optimal state assignment of FSMs (both synchronous and asynchronous) based on imposing adjacency relations among the states of the FSM (to insure minimal cost of the implementation and/or timing correctness of the transitions) led to the problem of embedding a suitably defined state adjacency graph into hypercubes [Arm62], [Sau72], [MSV83] . Other problems of software/hardware design led to the same problem, as in the case of finite automata modeling communication processes , with each state standing for a different process and state transitions resulting from atomic actions, performed by any process . Atomicity of actions can be guaranteed by assigning a bit vector of some fixed length to each state of the automaton (each position of the vector is a boolean variable that can be changed by any atomic action) in a way that adjacent states differ only in one position . This requires that the underlying graph of the automaton be cubical, i.e. a subgraph of some hypercube [APP85] .

It is worth at this point reviewing what is known about cubical graphs . We remember that an n-dimensional hypercube (that we will call hypercube or n-cube or simply cube, from now on) is seen as a graph Q_n with nodes the elements of $\{0,1\}^n$ and an edge between two nodes whenever the Hamming distance of the corresponding bit vectors is one .

More formally : The undirected graph Q_n and the directed graph DQ_n (called dicube) of the n-dimensional cube are defined as

$$V(Q_n) = V(DQ_n) = \left\{ u = (u_1,\ldots,u_n) : u_i \in \left\{0,1\right\}, i = 1,\ldots,n \right\}$$

$$E(Q_n) = \left\{ (u,v) : u,v \in V(Q_n) \text{ and } u,v \textit{ differ in exactly one coordinate} \right\}$$

the edge (u,v) of $V(Q_n)$ being in DQ_n directed from u to v iff the number of ones in u is less by one that that in v . DQ_n is an acyclic digraph having 2^n vertices and $n\,2^{(n-1)}$ arcs . For u in $V(Q_n)$, $u = (u_1,\ldots,u_n)$, the number $u_1+\ldots+u_n$ will be called the norm of u . A digraph DG is called cubical if there is n such that DG is isomorphic to a subgraph of DQ_n : the smallest n with this property is denoted $\dim(DG)$ ($\dim(G)$ is defined analogously in the undirected case, if graph G is cubical, then the dimension of G is the smallest n such that G is a subgraph of Q_n) .

The following equivalent definition has a more set-theoretic flavor [GaG73] and is especially suggestive of an underlying order relation. For a set S , define a graph $Q(S)$ called the cube on S, as follows . The vertices of $Q(S)$ are the finite subsets of S: the pair of subsets $\{S_1,S_2\}$ is an edge of $Q(S)$ iff the symmetric difference of S_1 and S_2 consists of a single element , i.e. $|S_1 \Delta S_2| = 1$. To each $T \subseteq S$, one can associate the characteristic function $\chi_T : S \rightarrow \left\{0,1\right\}$. For a finite set $S_n = \left\{s_1,s_2,\ldots,s_n\right\}$, χ can be used to coordinate $Q(S_n)$ by assigning to each $T \subseteq S_n$, the binary n-tuple $A(T)=(a_1,\ldots,a_n)$ where $a_k=1$ iff $s_k \in T$. An embedding of a graph G into $Q(S_n)$ is an injective mapping of the vertices of G into the vertices of $Q(S_n)$ which maps the edges of G into edges of $Q(S_n)$.

By the second definition, the n-cube can be considered as a poset with the natural partial ordering defined by

$$f \leqslant g \text{ iff } f(i) \leqslant g(i) \;\forall\; i \leqslant n.$$

The n-cube as a partially ordered set (poset) is isomorphic to the poset consisting of all subsets of an n-element set ordered by inclusion [Tro75] .

Cubical graphs are hard to characterize . Notice that the obvious property that a cubical graph has to be bipartite , is not sufficient, as $K_{2,3}$ (the complete bipartite graph with $|V_1| = 2$ and $|V_2| = 3$) shows . A first nice result by Djokovic [Djo73] characterized a subclass of them : those graphs that can be isometrically embedded in hypercubes, showing that G can be embedded in a cube so that distance is preserved iff G is bipartite and satisfies the following condition:

$G(a,b)$ is closed whenever a and b are adjacent vertices in G, where, $G(a,b)=\{v \in V(G) : d(v,a) > d(v,b)\}$, and $W \subset V(G)$ is closed if, for all $a,b \in W$ and $v \in V(G)$

$$d(a,v)+d(v,b) = d(a,b) \longrightarrow (implies)\ v \in W.$$

The two previous conditions are sufficient for a graph to be cubical. While bipartiteness is also necessary, the other condition is not as the figure 2.1 shows . Remember that if, for embedding $f : G \rightarrow Q(n)$, it is true that

$$d_G(v_1,v_2) = d_{Q(n)}(f(v_1),f(v_2))$$

then we say that f is an isometric embedding.

Havel and Liebl studied the embedding of trees in cubes [HaL72]. Havel and Moravel [HaM72] found an interesting property that cubical graphs inherit from cubes and is sufficient to characterize them : a graph is cubical iff it has a proper edge coloring. We recall that an edge coloring of a graph G is an assignment of colors to the edges of G such that no two adjacent edges have the same color. A path is a connected set of edges with maximum vertex degree two . If all nodes have degree two, the path is called a cycle . Given an edge coloring and a path, we call the path even if all colors appear an even number of times in the path. Finally an edge coloring is proper when a path is even with respect to it iff it is a cycle . A corollary is that a graph is cubical iff all its biconnected components are .

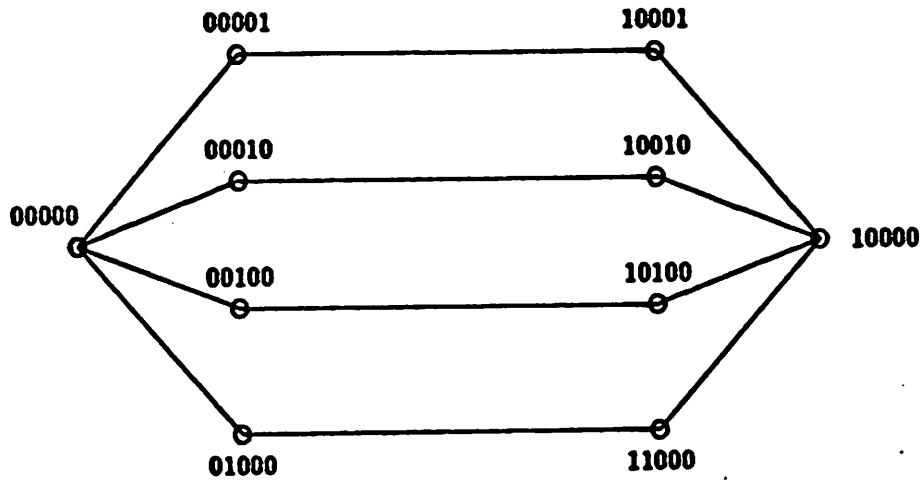


Fig. 2.1. A cubical graph with no isometric embedding

Garey and Graham [GaG73] showed that any cubical graph has dimension at most $|V|-1$. Afrati, Papadimitriou and Papageorgiou [APP85] improved the bound for biconnected graphs to at most lower ceiling of $|V|/2$, deducing as a corollary that the dimension of a graph with n nodes and k biconnected components is at most $(n+k-1)/2$.

They established the computational complexity of the problem CUBICAL GRAPHS: given a graph, is it cubical ? showing that it is NP-complete , by reducing EXACT COVER to it .

What can be said about the dimension of specific classes of graphs ? Unfortunately, it is even hard to prove things for trees (which are cubical, because they have no cycles and each edge can be given a different color) . Havel and Liebl [HaL72] proved that a full binary tree with height h has dimension $h+2$. While full binary trees can be embedded in low-dimensional cubes , trees like the star attain the maximum possible value ($|V|-1$) . There is an exponential gap between the lower and upper bounds . Afrati et al. [APP85] suggested an heuristics that uses at most k^2 colors when applied to a tree of dimension k (and can be improved down to $k^2 / \log k$) and conjectured that calculating the dimension of a tree is NP-complete .

2.3. How to cope with the unpleasant realities of cubical graphs

Two approaches were taken by researchers to embed in a cube graphs that are not cubical : "relax" the graph or "relax" the cube . An example of the first kind was proposed by Hechler and Kainen [HeK74], who investigated the conditions for the embeddability of a digraph in a dicube . They introduced the concept of subdivision to relax a given digraph and allow its embedding in a cube and gave conditions on the subdivisions of a graph to ensure embeddability . If D is a digraph, we call the graph D' a subdivision if it is obtained by replacing certain arcs a by directed paths all of whose interior points are new and which join the points originally joined by a .

Armstrong [Arm62] and Saucier [Sau72] represented the state assignment problem as a subgraph isomorphism problem, where a one-to-one mapping is sought between the set of the vertices of the adjacency graph (states) and a subset of the vertices of a given boolean hypercube (codes) . Since such an isomorphism may not exist, Armstrong and Saucier relaxed some adjacency requirements, although their technique even by increasing the code length (augmenting the dimension of the hypercube) does

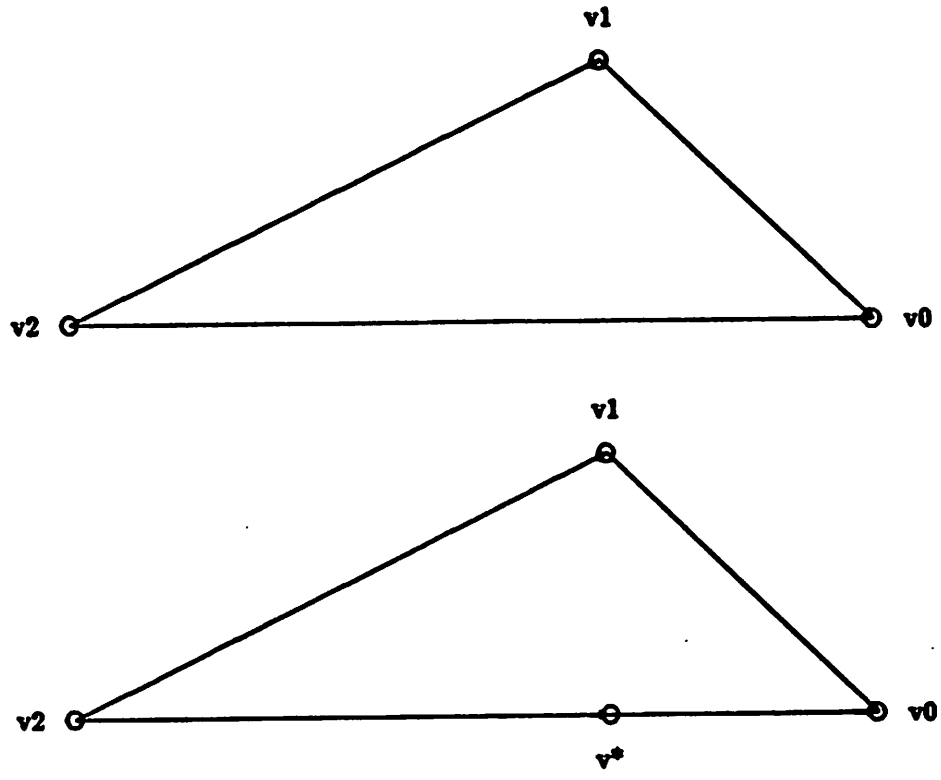


Fig. 2.2. The upper triangle has a subdivision (below) that embeds in Q_2

not guarantee that a distance-preserving embedding exists .

Graham and Pollack [GrP71], [GrP72], working in the undirected case, modified the cube ("squashed cube") instead of the graph . They considered the following problem : assign a k-digit binary "address" to each vertex v of a graph G in such a way that $d(v,w)=d(a(v),a(w))$, where $d(v,w)$ denotes the distance from v to w in G and $d(a(v),a(w))$ is the Hamming distance from $a(v)$ to $a(w)$, i.e. the number of coordinates at which $a(v)$ and $a(w)$ differ . Such an addressing corresponds to an embedding of G in Q_k . But, this in not possible in general: for example K_3 cannot be

addressed in this way . They insured that a solution could always be found by introducing the don't care besides 0 and 1 in the addresses, with distance computed by counting two coordinates as distinct only when one has a 1 and other has a 0 . This amounts to embedding G in a cube some of whose faces have been "squashed" . Moreover, the dimension of the cube is at most $s(n-1)$ where s is the maximum distance between any two vertices of G , and an algorithm was given to obtain the cube which always produced a cube of dimension at most $n-1$. Yao [Yao78] proved that any graph with n vertices has such an addressing scheme of length bounded by $nlogn$.

A program for optimal state assignment developed at UCB in the 1983 [MSV83] considered the problem of assigning codes satisfying distance relations . Adjacency code assignment was represented as an embedding of an adjacency graph into squashed cubes . The results were not completely satisfactory because it wasn't easy to effectively bound the code-length and especially because this wasn't the most appropriate model for the cost function involved in the problem .

2.4. Our model

Cubical graphs, although deceivingly close to what required by CONSTRAINED CUBICAL EMBEDDING, are not really what we should be concerned with when solving that problem . In a sense the definition of an n-cube given in section 2.2 suggests the wrong structure underlying the cube, or more properly the wrong cube, because it doesn't capture what we care most in CONSTRAINED CUBICAL EMBEDDING, i.e. the representation of the inclusion relations among the faces of the hypercube . Indeed, we want to group together the state codes in binary-valued logical implicants in the same way states are grouped together in the minimal symbolic (multiple-valued) cover , to obtain a binary-valued cover of the FSM combinational component having as many

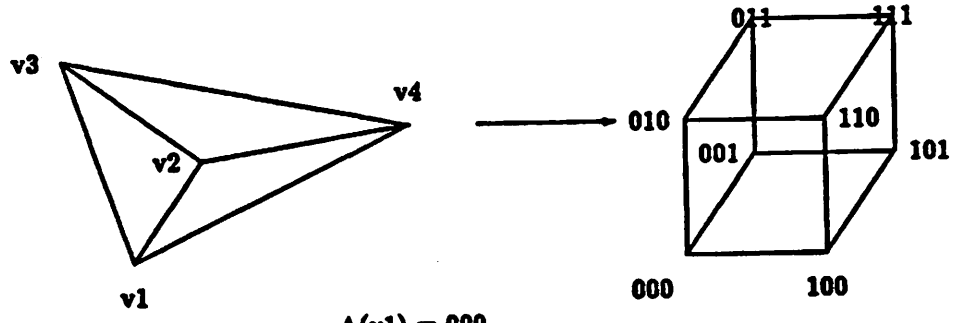


Fig. 2.3. Squashing a 3-cube to embed a graph

implicants as the minimal symbolic cover . An encoding such that each subcube contains all and only the codes of the states included in the corresponding subset of states satisfies the above requirements . In fact, each coded implicant represents all and only the state transitions related to the corresponding symbolic implicant . The pitfall we warned about, wasn't avoided in [MSV83]: later approaches [DeM83], [Vil86c] realized that, but renounced to analyzing the combinatorial structure of the problem and implemented reasonable heuristics, based mostly on the geometric intuition . Since it is more likely that sound heuristics arise from the "right" model of the problem, we will reformulate CONSTRAINED CUBICAL EMBEDDING in a proper way .

Since, for our purposes, it is essential to capture the inclusion relations among the faces of the hypercube, we choose to represent it with its underlying face-poset , obtained by ordering all faces of all available dimensions according to the boolean inclusion relation .

Formally the n -cube face-poset (or n -face-poset) is the set of all sequences of 0, 1, x (don't care) of length n (called faces) . It is a poset with the natural partial ordering defined by $f \Leftarrow g$ iff $f(i) \leqslant g(i)$ for all $i \leqslant n$ (note that $0 \leqslant x$, $1 \leqslant x$) . (the face-poset is completely different from the poset structure induced on the n -cube by the natural partial ordering on the Hamming codes of the vertices and isomorphic to the poset consisting of all subsets of an n -element set ordered by inclusion) . Level of a face is the number of x's contained in the sequence . There are $n+1$ levels in a n -face-poset . Cardinality or dimension of a face is $2^{(level\_of\_the\_face)}$. We define the intersection of faces, with the usual boolean rules .

Proposition 2.1 The number of faces of an n -face-poset is 3^n .

Proof . For each level $i = 1,....,n$ there are $\binom{n}{i}$ ways of choosing the care positions and each choice generates 2^i different faces . Summing up over all levels and applying the binomial theorem, we obtain 3^n .

■

It helps the intuition to draw the posets as Hasse diagrams .

Also the collection of constraint relations of the problem instance can be seen as a poset by ordering them according to the set inclusion relation .

Within this setting CONSTRAINED CUBICAL EMBEDDING may be modeled by :

SUBPOSET DIMENSION :

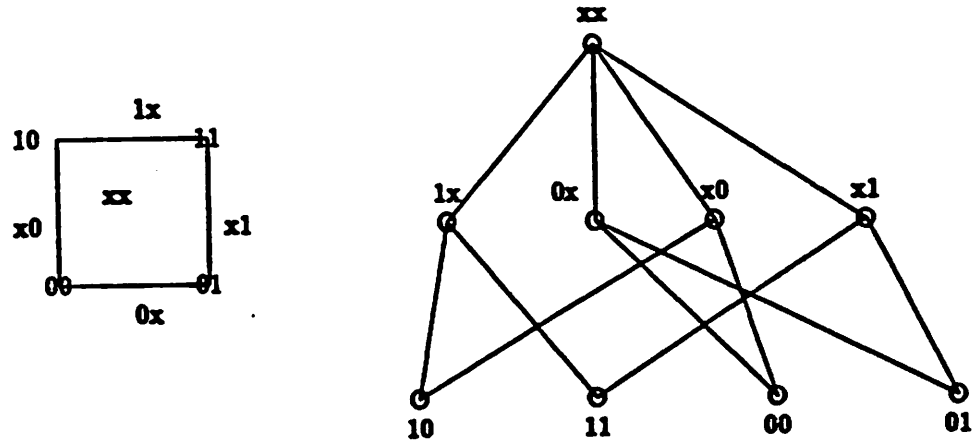


Fig. 2.4. A 2-face-poset

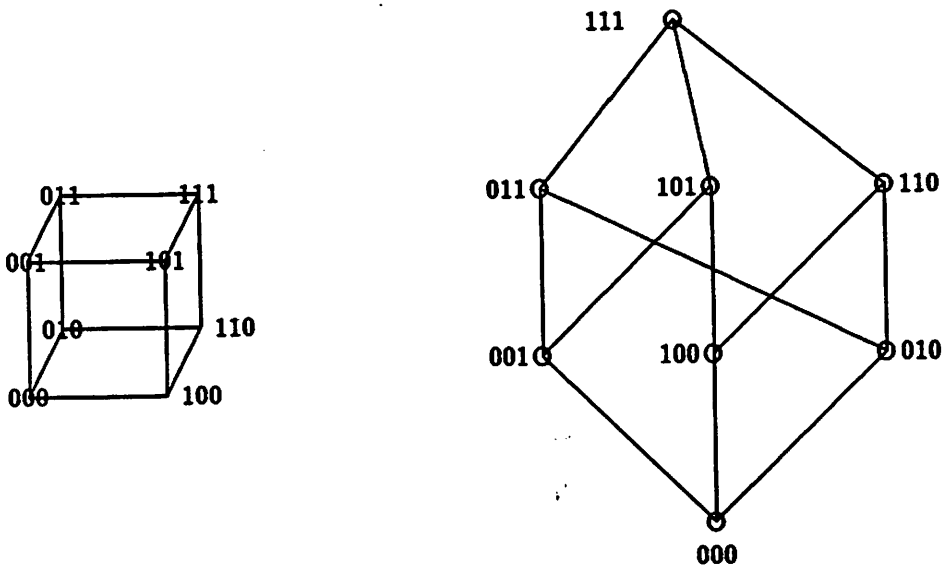


Fig. 2.5. A 3-cube

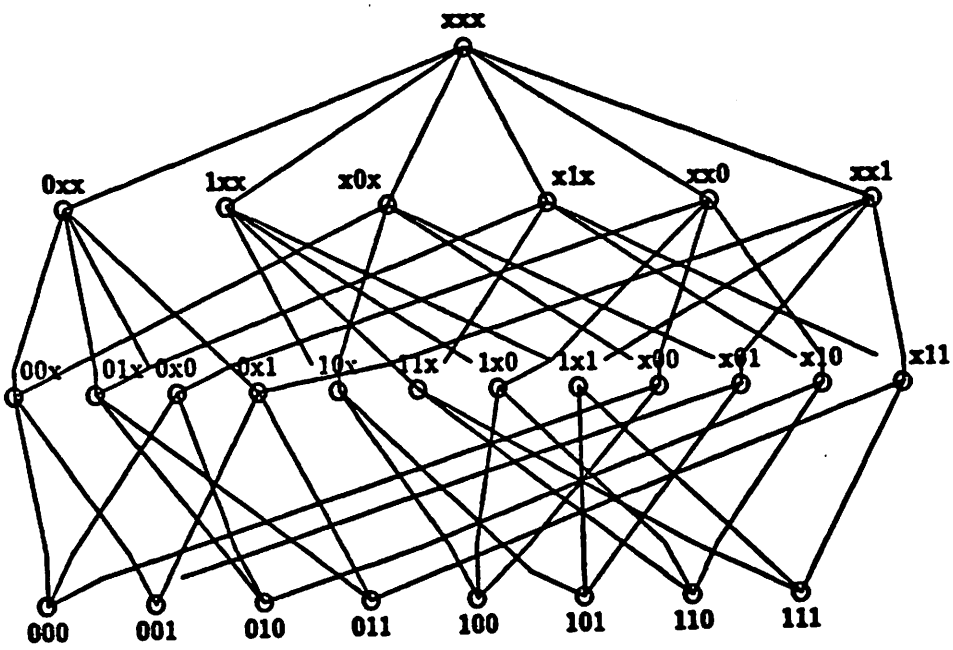


Fig. 2.6. A 3-face-poset

INSTANCE : Set $S = \{1,...,n\}$ and a collection $C(S_i)$ of subsets $S_i \subseteq S$. $i = 1,...,m$.

QUESTION : find the minimum k and an injective map f from the sets $\in$ *Closure* $_{\cap}[C(S_i)]$ (where

$$\textit{Closure}_{\cap}[C(S_i)] = C(S_i) \bigcup \left\{ S_j \ : \ S_j = S_{j1} \bigcap S_{j2} \ . \ S_{j1} \ . \ S_{j2} \in C(S_i) \right\}$$

and $\bigcap$. $\bigcup$ are assumed in set-theoretical sense) to the faces of the k-cube

$f : S_i \rightarrow f(S_i)$

satisfying cardinality(S_i) <= cardinality($f(S_i)$) and such that the k-cube contains a poset equivalent to the given one . i.e. for all subsets S_i . S_j . S_k $\in$ *Closure* $_{\cap}[C(S_i)]$:

$$S_i \supset S_j \text{ iff } f(S_i) \supset f(S_j)$$

[f preserves inclusions] :

$$S_i \bigcap S_j = S_k \text{ iff } f(S_i) \bigcap f(S_j) = f(S_k)$$

[f preserves intersections] .

SUBPOSET EQUIVALENCE :

INSTANCE : Set $S = \{1,....n\}$ and a collection $C(S_i)$ of subsets $S_i \subseteq S$, $i = 1....m$, and a positive integer k .

QUESTION : does the k-cube contain a poset equivalent to the given one, i.e. is there an injective map f from the sets $\in$ *Closure* $_{\bigcap}[C(S_i)]$ (where

$$\textit{Closure}_{\bigcap}[C(S_i)] = C(S_i) \bigcup \left\{ S_j : S_j = S_{j1} \bigcap S_{j2} , \ S_{j1} , S_{j2} \in C(S_i) \right\}$$

and $\bigcap$, $\bigcup$ are assumed in set-theoretical sense) to the faces of the k-cube

$$f : S_i \rightarrow f(S_i)$$

satisfying cardinality(S_i) <= cardinality($f(S_i)$) and such that for all subsets $S_i , S_j , S_k \in$ *Closure* $_{\bigcap}[C(S_i)]$:

$$S_i \supset S_j \text{ iff } f(S_i) \supset f(S_j)$$

[f preserves inclusions] ,

$$S_i \bigcap S_j = S_k \text{ iff } f(S_i) \bigcap f(S_j) = f(S_k)$$

[f preserves intersections] ?

Proposition 2.2 SUBPOSET EQUIVALENCE is NP-complete .

Proof . Consider the Hasse diagram of *Closure* $_{\bigcap}[C(S_i)] \bigcup \left\{ S \right\}$. We say that it is *completely leveled* if it has $k+1$ levels of inclusion and for every constraint S_i of cardinality c there are constraints S_j of cardinality $c/2$ such that

$$\bigcup S_j = S_i$$

Here, $\bigcup$ is assumed in set-theoretical sense (see Figure 2.7) . Restrict SUBPOSET

EQUIVALENCE to SUBGRAPH ISOMORPHISM by allowing only instances which are *completely leveled* . There is a similar restriction to SUBGRAPH HOMEOMORPHISM (see Figure 2.8) .

■

Proposition 2.3 SUBPOSET DIMENSION is NP-hard .

Proof . SUBPOSET EQUIVALENCE, the recognition version is NP-complete and is no harder than the optimization problem . NP-hard means that there is no succinct certificate for an answer other than verifying all possible assignments in cubes of smaller dimensions than the minimum one given in the answer .

■

Remark: it could have been also interesting to couche the problem in terms of hypergraph embedding . Both the constraints of the instance of the problem and the faces of the cube can be seen as collection of subsets defining the hypergraphs of a subhypergraph isomorphism problem . We have not looked yet in that direction . while we will pursue in the next chapter an algorithmic solution to SUBPOSET DIMENSION .

2.5. Implicit representation of graphs : labeling schemes and universal graphs

We notice an interesting analogy with recent research [KNR87] on implicit representation of graphs . In the usual representation of an n -node graph, the names of the nodes (i.e. the integers from 1 to n) betray nothing about the graph itself . The names (or labels) of the n nodes are just $\log n$ bit place holders to allow data on the edges to code for the structure of the graph . It would be better to be able, by

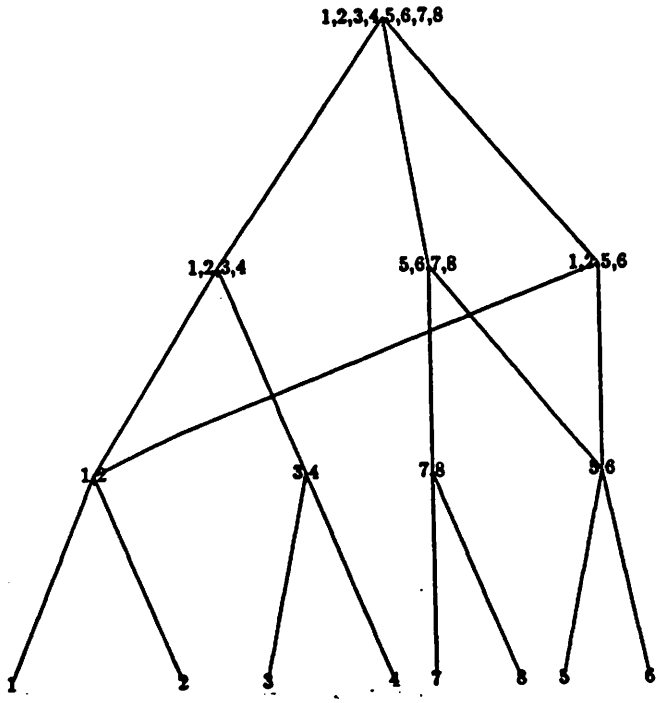


Fig. 2.7(a). Completely leveled instance (restriction to SUBGRAPH ISOMORPHISM)

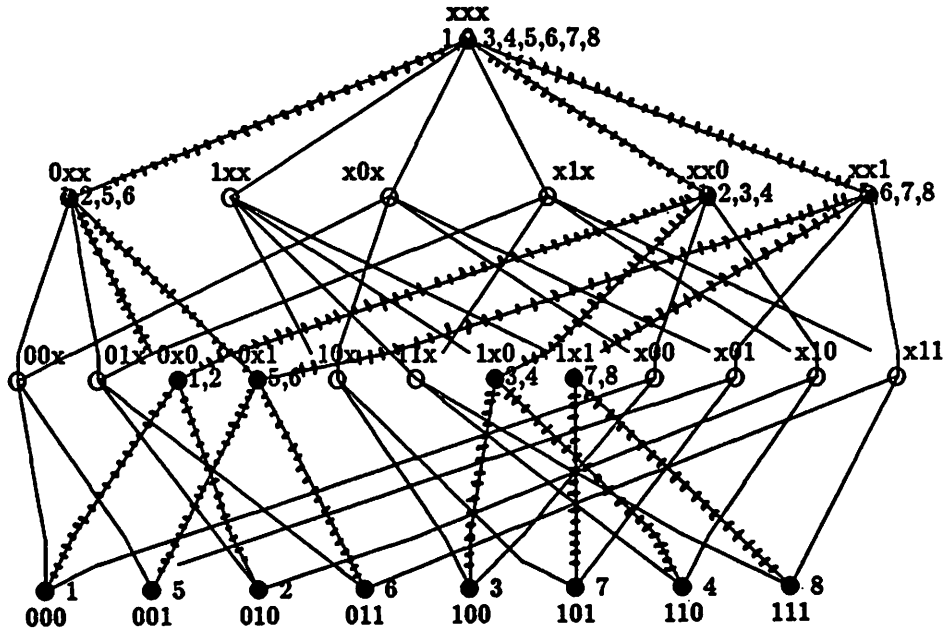


Fig. 2.7(b). 3-face-poset with subgraph isomorphic to instance poset of Fig. 2.7(a).

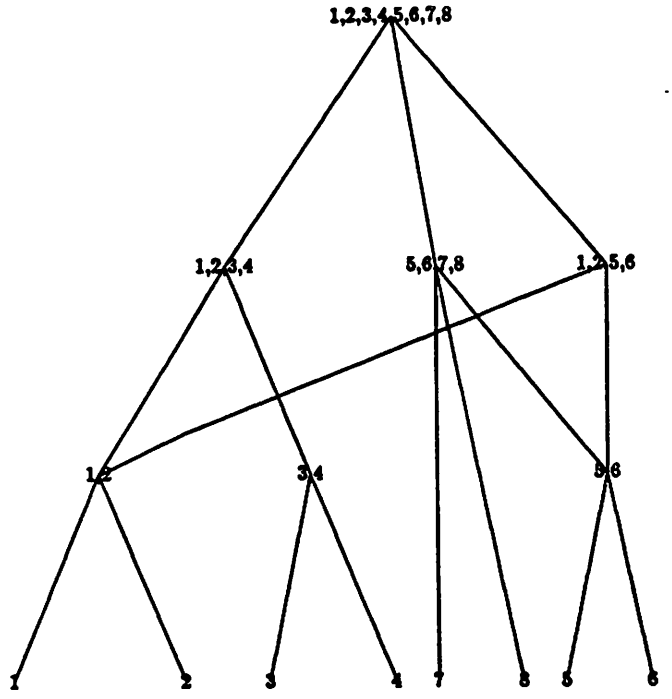


Fig. 2.8(a). Partially leveled instance (restriction to SUBGRAPH HOMEOMORPHISM)

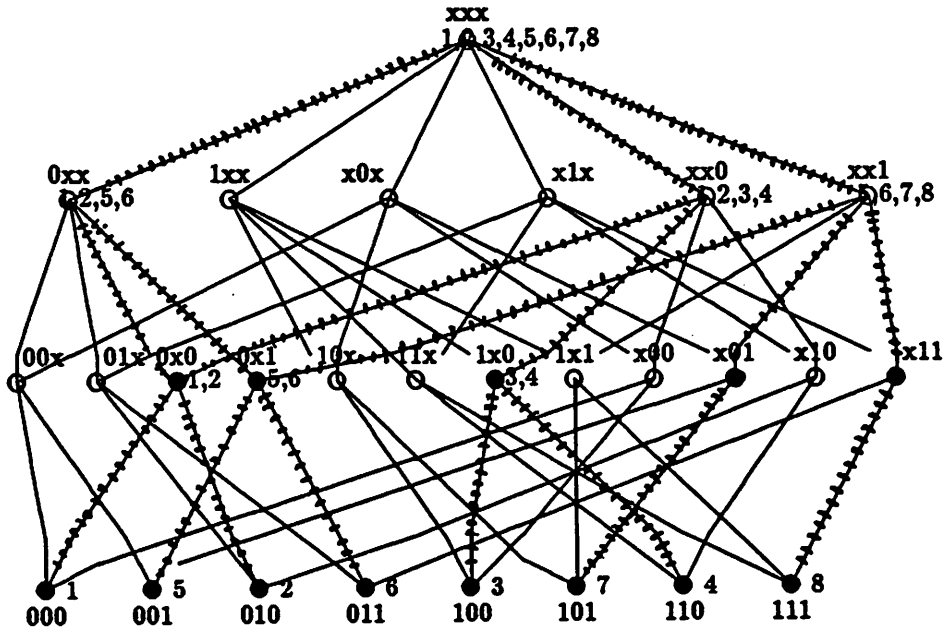


Fig. 2.8(b). 3-face-poset with subgraph homeomorphic to instance poset of Fig. 2.8.(a).

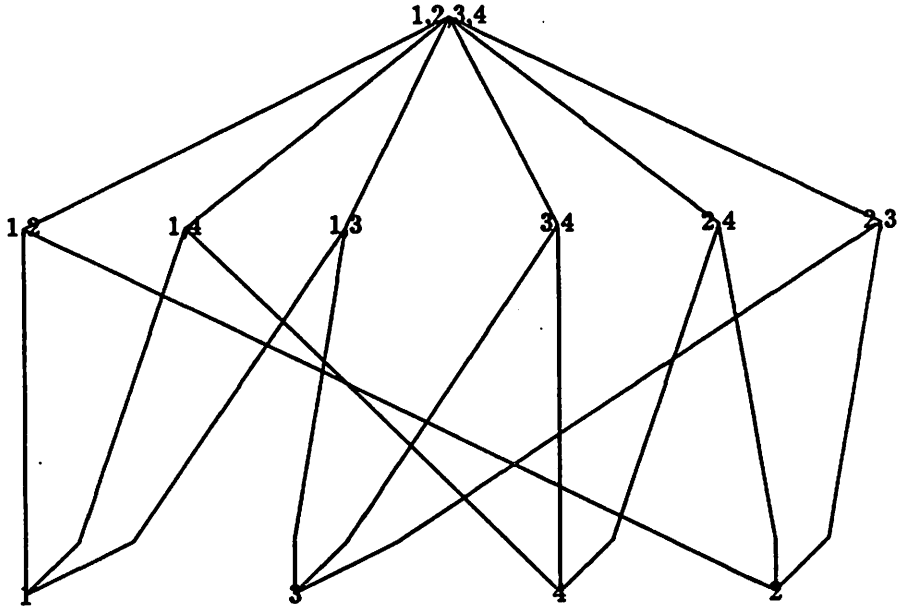


Fig. 2.9(a). General instance

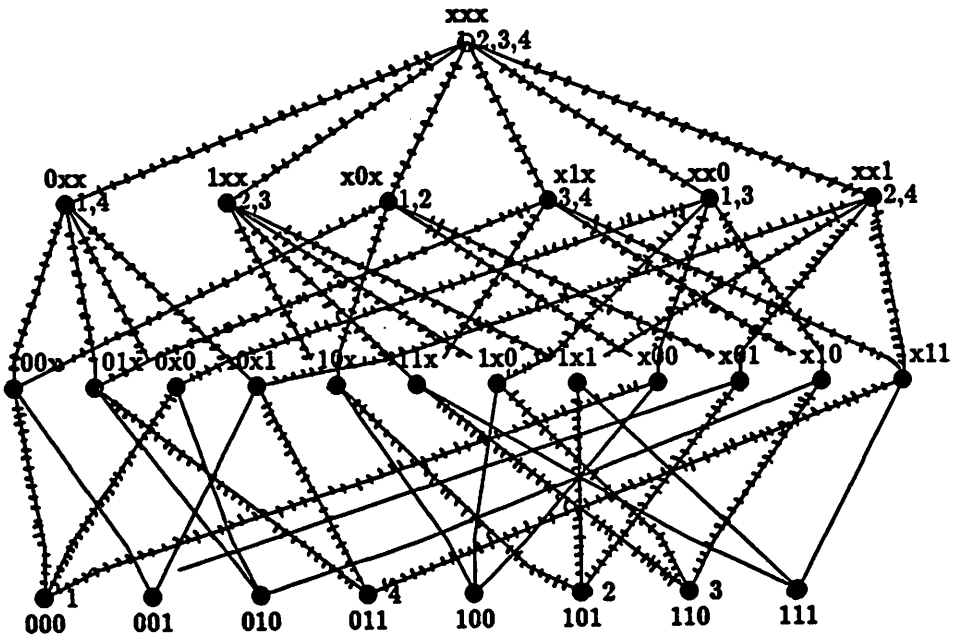


Fig. 2.9(b). 3-face-poset with subposet equivalent to instance poset of Fig. 2.9(a).

assigning $O(logn)$ bit labels to the nodes, to code completely the structure of the graph, so that given the labels of two nodes one can test if they are adjacent in time linear in the size of the labels .

More formally a family F of finite graphs has a k-labeling scheme if there is a polynomial time Turing machine T, and a function f which labels the nodes of each graph G in F with distinct labels of no more than $klogn$ bits (n is the size of G), such that given two node labels of a graph G in F, T will correctly decide adjacency of the corresponding nodes of G . If a family F has a k-labeling scheme for some k, we say that it has a labeling scheme .

A related notion is that of universal graphs . Formally, given a finite set of graphs S, a graph G is universal for S, if every graph in S is a vertex induced subgraph of G. A family has universal graphs of size $g(n)$, if for every n, there is a graph of size less than $g(n)$, which is universal for the set of all graphs in F with fewer than n nodes . Labeling schemes and universal graphs are closely related, in the sense that if a family F has a k-labeling scheme, then it has universal graphs of size n^k constructible in logspace.

The labels on the nodes of the universal graph make the embedding of a graph G in its universal graph easy to find . To embed a graph G in its universal graph, all that is required is to label G. The labels then give information about the embedding. An answer to SUBPOSET DIMENSION essentially is a generalized labeling of a given graph on the class of universal graphs of the n-face-posets for minimum n .

CHAPTER 3

An algorithm for solving exactly SUBPOSET DIMENSION

In this chapter we will describe an algorithm to solve exactly SUBPOSET DIMENSION . It is already clear from the computational analysis given in Chapter 2, that in the worst case an exact solution is hopelessly exponential . But we pursue it, because we need a tool for the validation of approximate solutions to CONSTRAINED CUBICAL EMBEDDING based on heuristics . The approaches taken so far in other programs are :

(i) either satisfy all constraints (KISS), and try to keep the dimension as close as possible to the minimum(unknown) one while assigning constraints to faces in a heuristic fashion in general much more wasteful of space than the exact solution;

(ii) or limit the dimension of the cube to a fixed value (for instance in NOVA the log of the cardinality of the universe of the states), and try to satisfy as many constraints as possible within the given boolean space .

The problems with (i) are that without being able to compute an exact embedding it is not possible to validate the quality of the proposed heuristics ; given that our experimental results show poor gains in area even with an exact embedding when the minimum dimension grows too much with respect to the information-theoretic minimum, algorithms that look for suboptimal satisfaction of all constraints have to be tested against exact solutions, to understand what at most we can hope from them .

The problems with (ii) are that it is hard to define quantitatively the meaning of satisfying "as many constraints as possible", without a precise measure of what we gain/loose, in terms of area, adding/dropping a constraint . With the availability of a program for the exact solution, we may come up with a good evaluation of the "merit"

of each constraint in the exact case (i.e. final area satisfying the complete given poset / final area satisfying the given poset without the constraint to evaluate) to suggest right heuristics on dropping constraints, when in need to trim a given poset to fit in the available boolean space .

3.1. Processing a problem instance of SUBPOSET DIMENSION

The poset of the constraints of an instance of SUBPOSET DIMENSION is represented by its Hasse diagram, i.e. an acyclic directed graph whose nodes are the constraints and an arc goes from v_i to v_j when subset S_i includes subset S_j . We call it the poset graph or instance poset (to distinguish it from the other poset involved, the n-face-poset) . More constraints are added to the poset graph to make all the inclusion relations already implicit in it explicit . Namely, not already present constraints that may be obtained as intersections of other constraints (and related arcs) are added, because any assignment satisfying the original constraints must satisfy their intersections too, and therefore, in designing a constructive algorithm, it pays to see the poset structure completely unfolded . We add to the poset graph also the following trivial constraints (and related arcs), if they are not already there :

1) the constraint representing the universe;

2) the constraints representing the singletons .

The reason, again, is to make sure that the poset graph contains all the inclusion relations of the problem . Especially, we add the singletons because their assignment (consistent with that of all subsets including them, directly and undirectly), which is the ultimate goal of our embedding, has to be enforced explicitly (if they are not already part of the problem instance) .

Lastly, to every constraint c in the poset graph, we associate the set of his fathers $F(c)$ and the set of his children $C(c)$, where

$F(c)$ = { set of constraints that include properly c and don't contain properly any other constraint that includes properly c }

in other words, they are the minimal constraints that include c

$C(c)$ = { set of constraints included properly in c and not included properly in any other constraint included properly in c }

in other words, they are the maximal constraints included in c

We can easily recognize that the relations of fathers and children, are a coincise representation of the Hasse diagram arcs (instead of the arcs between all possible comparable constraints) . We walk through the poset graph from a constraint to another, through the fathers and the children of the constraint on which we are currently placed .

3.2. General structure of the exact encoding algorithm and upper level backtracking

The exact encoding algorithm that we implemented finds an answer to SUBPOSET DIMENSION, by answering exactly to SUBPOSET EQUIVALENCE on the range of feasible dimensions . SUBPOSET EQUIVALENCE asks if the instance poset can be embedded in a given k-face-poset (and finds a satisfactory assignment, when it exists) . If it is so for dimension k and we already answered "no" for the feasible dimensions $< k$, we have an answer to SUBPOSET DIMENSION, i.e. k is the minimum dimension and the sought assignment is returned by SUBPOSET EQUIVALENCE . We know that the algorithm will not invoke SUBSET EQUIVALENCE on face-posets of increasing dimensions endlessly, but that it will come up eventually with a "yes", because of a well-known upper bound on SUBPOSET DIMENSION : any instance poset can be

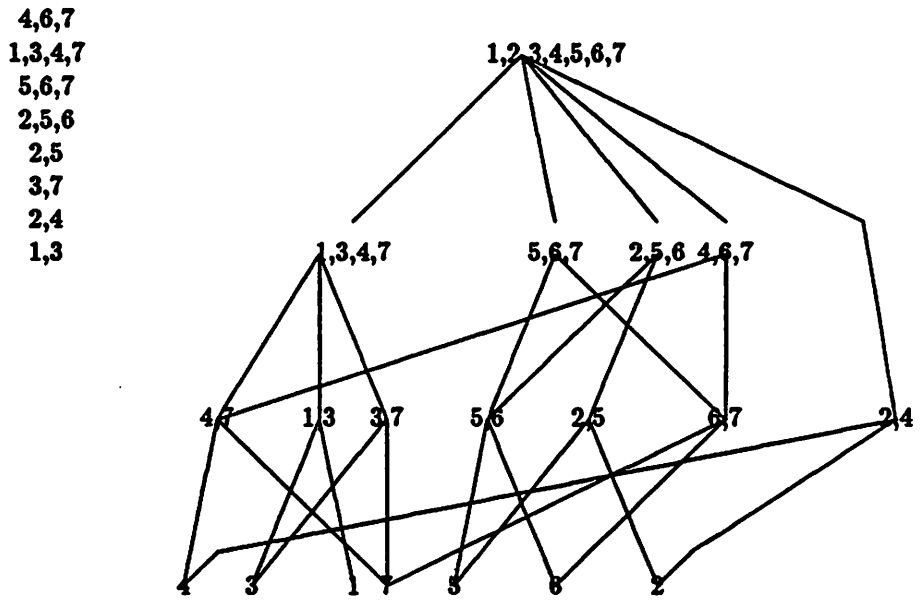


Fig. 3.1. Example of instance constraints and instance poset

embedded in at most an n-face-poset if its universe has cardinality n .

At the other end of the range of feasible dimensions, we need lower bounds to spare useless invocations of SUBPOSET EQUIVALENCE . The most trivial one is that the k-face-poset should have at least log $|S|$ (log of the cardinality of the universe) 0-dimensional faces; it can be generalized to the criterion that the k-face-poset should have at least as many faces of a given cardinality as the instance poset has constraints of a given cardinality (counting necessary condition) . It would be important to come up with more powerful necessary conditions that need to be satisfied by a solution, to allow a quick rejection of dimensions unfeasible for a given problem instance . This is

still an open area of investigation in the future developments of the algorithm . The current version calls a routine *least_dimcube* that implements the counting necessary condition and returns the lower bound from which to start the invocations of SUBPOSET EQUIVALENCE .

The main routine is based on a two-tier backtracking cycle, called respectively *backtrack_up* and *backtrack_down* . It is no wonder that some backtracking is involved to find an exact solution, but to explain the nested backtracking mechanism implemented, we need to introduce the concept of cardinality configuration . According to the statement of SUBPOSET DIMENSION a feasible assignment between constraint S_i of the instance poset and face $f(S_i)$ of the face-poset needs to satisfy the cardinality hypothesis : cardinality(S_i)<=cardinality($f(S_i)$) . Given the current dimension of the hypercube on which we are invoking SUBPOSET EQUIVALENCE, sometimes a solution exists only when the previous inequality is proper for one or more of the constraints . The reason may be that we are dealing with constraints that are not a power of two and freeze more space on the hypercube than their cardinality would tell and/or constraints that need to be adjacent to many others and so require a large face (which means a large boundary) to satisfy their numerous intersections with others . So, for a fixed embedding dimension, it looks like that we have for every constraint not only the choice of many possible faces of its given cardinality, but even the choice of many possible cardinalities of the face to which it can be assigned . This sounds frightening from a computational point of view . Fortunately, not all constraints enjoy this high degree of freedom .

We classified constraints in 4 categories :

category 0) the universe;

category 1) their only father is the universe (called primary constraints);

category 2) they have at least two fathers;

category 3) their only father is not the universe.

The universe has only a possible assignment . Constraints of category 2) are completely determined (not only in terms of cardinality) by their fathers, because they get as face the intersection of the faces assigned to their fathers; constraints of cardinality 3) enjoy the freedom that their only father allows them, but since their father is not the universe, it is unlikely that the cardinality of the face to which they can be assigned will span a large range; only constraints of cardinality 1) enjoy a high degree of freedom, because their image cardinality spans the entire range from the minimum to that of the universe - 1 . To take this into account, a routine called *faces_dim_set* returns at every call a new feasible cardinality configuration, mapping each constraint of category 1 into a feasible face cardinality; an upper level backtrack mechanism will call *faces_dim_set* every time that the lower level embedding mechanism (the second backtrack, that we will describe later) isn't able to find an encoding with the given cardinality configuration (and given hypercube dimension). When, for a given embedding dimension, all possible cardinality configurations have been unsuccessfully tried, the main routine updates the hypercube dimension to a dimension larger by one than the current one .

We are aware of how important a smart upper level backtracking mechanism is, for the computational efficiency of the algorithm, since every new iteration through it may be an expensive computational trigger of computations at the lower level . The one currently implemented is simple, because the choices are still under evaluation at the moment of the present writing . It must be said that one needs to experiment with rather large examples to test meaningfully different heuristics at the upper backtracking level .

The heuristic currently implemented generates the cardinality configurations in increasing order of magnitude, i.e., it first tries to assign to a constraint the minimum

face cardinality larger than or equal to its own cardinality and later increases it; what is hard to figure out is the interactions among the choices for the different primary constraints and also necessary conditions that can trim as much as possible the space of the possible cardinality configurations . We plan to look carefully into this issue .

As a summary, we offer a complete view of the external cycle of the exact algorithm

```
{
   code_found = FALSE;
   for (dim_cube = least_dimcube(|S|); dim_cube <= |S|; dim_cube++) {
        /* backtrack_up finds a feasible cardinality configuration
        of faces ( it calls faces_dim_set(|S|,dim_cube) )        */
        outer_loop = backtrack_up(|S|,dim_cube);
        while (code_found == FALSE && outer_loop == TRUE) {
           /* backtrack_down finds if an encoding exist given the
           current cardinality configuration and cube dimension */
         code_found = backtrack_down(|S|,dim_cube);
           /* finds a new feasible cardinality configuration */
         outer_loop = backtrack_up(|S|,dim_cube);
        }
        if (code_found == TRUE) {
           break;
      }
```

```
    }

    if (code_found == FALSE) {
        something went wrong : upper bound was overflowed
    }

}
```

3.3. Lower level backtracking

Lower level backtracking (*backtrack_down*) answers SUBPOSET EQUIVALENCE, given an hypercube dimension and a cardinality configuration . The main tasks that need to be carried out at this level are 1) walking through the instance poset and the face-poset, and 2) mapping constraints of the first to faces of the second . An assignment is built incrementally and when it cannot be extended, because of previous wrong choices, *backtrack_down* undoes part of later work and new maps from constraints to faces are attempted from a previous stage . In case no feasible assignment exists, *backtrack_down* answers "no" to SUBPOSET EQUIVALENCE . It is crucial to design efficiently backtrack_down, because it is the basic computational device of the exact encoding algorithm .

3.3.1. Walking through the instance poset

One can think of many ways of picking the constraints (i.e. introducing among them an encoding order) to assign them a face : choosing many of them at a time or just one at a time . What matters mostly is to choose them in a way such that it can be discovered as soon as possible that a given partial assignment is unfeasible, i.e. it can-

not be extended to a complete assignment; especially crucial is the capability of discovering quickly that no assignment is feasible with the given space parameters . The property of an assignment of being feasible is global (it depends on all the inclusion and intersection relations among constraints), while we can treat fairly well (especially with a sequential algorithm) local information , so we must trade-off between making the selection far-sighted enough and keeping the building of the assignment reasonably local .

Our heuristic orders the constraints in order of decreasing feasible face cardinality (encoding level) and, the encoding level being the same, selects first the constraints sharing sons with already assigned constraints . The rationale is that we want to code first the constraints needing larger faces and that we exploit a look-ahead of one level (sharing sons) to reject encodings at the upper level if they are unable to satisfy intersections at the next lower level . This allows to discover at the upmost possible level when an assignment is unfeasible, i.e. cannot be extended downwards . Constraints of category 2 (with at least two fathers) are not seen by the select routine because their code is decided by the encoding of their fathers, which happens before them . Only constraints of category 1 and 3 really matter, and, since the latter's faces must be contained in the faces assigned to their fathers, only the former enjoy true freedom ; so we select them soon and build a feasible assignment, while taking into account also the intersections at one level down . If no such assignment exists we are able to discover it quickly, in the worst-case we may assign to most constraints of category 1 all feasible faces, but we strongly bound the backtracking with faces of lower encoding level . One could extend the look-ahead to two or more encoding levels down, and we plan to experiment with it in the future . Up to now, we got good results with only one level of look-ahead . The choice of the first constraint may be tuned a little more, up to now we didn't have reasons to favor more complex criteria .

Summary of the structure of the selection routine *next_to_code*

the first time :

return a constraint of category 1 (different from the universe)
with maximum feasible face cardinality (given by the cardinality
configuration mapping) .

the successive times :

```
if (there is an unassigned constraint "c" of the same feasible face
   cardinality sharing a son with the last chosen constraint) {
      return "c"
} else {
   if (there is unassigned constraint "c" of the same feasible face
        cardinality) {
         return "c"
   } else {
        if (there is un unassigned constraint "c" of maximum smaller
            feasible face cardinality) {
            return "c"
       } else {
            return none;
       }
   }
}
```

3.3.2. Overview of backtrack_down

It is now possible to offer a general perspective of *backtrack_down* . Three main routines are involved in the cycle :

1) *next_to_code* that walks through the instance poset and was examined in 3.3.1;

2) *assign_face* that walks through the face-poset and assigns faces to constraints and will be examined in 3.3.3;

3) *select_backtrack_constr* that implements the backtracking control mechanism, i.e. it undoes a partially realized assignment which cannot be extended or it recovers after previous wrong decisions have been corrected . Decisions are taken with help from the list of the already selected constraints of cardinality 1, inserted in the order in which they have been chosen by *next_to_code*; when facing an assignment which cannot be extended, the constraint inserted there as last and related (e.g., by inclusion or intersection) to the current unassignable constraint(s) is sent back to *assign_face* and part of the assignment is undone . We are currently tuning the choice of how much to undo, many times undoing only the last coded constraint seems to work fine .

We notice that the code given to the first chosen constraint, because of invariance under rotations and reflections, doesn't affect the feasibility of an assignment, in other words when *backtrack_down* comes back to the first code it means that no feasible assignment exists .

scheme of *backtrack_down*

```
{

  next_constr = next_to_code();
```

```
while (next_constr != (CONSTRAINT *) 0) {

   backtrack = FALSE;

     face_found = assign_face(next_constr);

   while ( ! (face_found == TRUE && backtrack == FALSE) ) {

       if (face_found == FALSE && backtrack == FALSE) {

          /* a backtracking phase starts */
        backtrack = TRUE;

          if (back to the first selected constraint) {
               /* no feasible assignment exist */
               return;
          } else {
               backtrack_constr = select_backtrack_constr();
             face_found = assign_face(backtrack_constr);
          }
       }

       if (face_found == FALSE && backtrack == TRUE) {

          /* the current backtracking phase continues */
          if (back to the first selected constraint) {
               /* no feasible assignment exist */
               return;
          } else {

               backtrack_constr = select_backtrack_constr();
```

```
            face_found = assign_face(backtrack_constr);
          }
        }

        if (face_found == TRUE && backtrack == TRUE) {

          /* a backtracking phase ends */
            backtrack_constr = select_backtrack_constr();
          face_found = assign_face(backtrack_constr);
          if (again to the last constraint selected by next_to_code) {
              backtrack = FALSE;
          }
        }

      }

      next_constr = next_to_code();

    }

}
```

3.3.3. Walking through the face-poset and assigning faces

A careful reader will have noticed that after having introduced the n-face-poset, we glissed over its huge size (3^n nodes for constraints drawn from an universe of cardinality n). Are we really going to build such a monster and walk through it ? Obviously no, this would destroy any hope of coming up with a solution efficient even on the average . Fortunately, the n-face-poset enjoys the enviable property of

possessing a natural labeling scheme, so that given the label of a face we can say immediately which are its neighbors, i.e. the nodes connected to it by ingoing or outgoing edges . Equally easy is to generate all faces of a given cardinality . So we will not build nor store in any way the n -face-poset, but instead a routine *gen_newface* will be able to return faces in the neighborhood of a given one, i.e. it will walk through the n -face-poset without explicitly generating it .

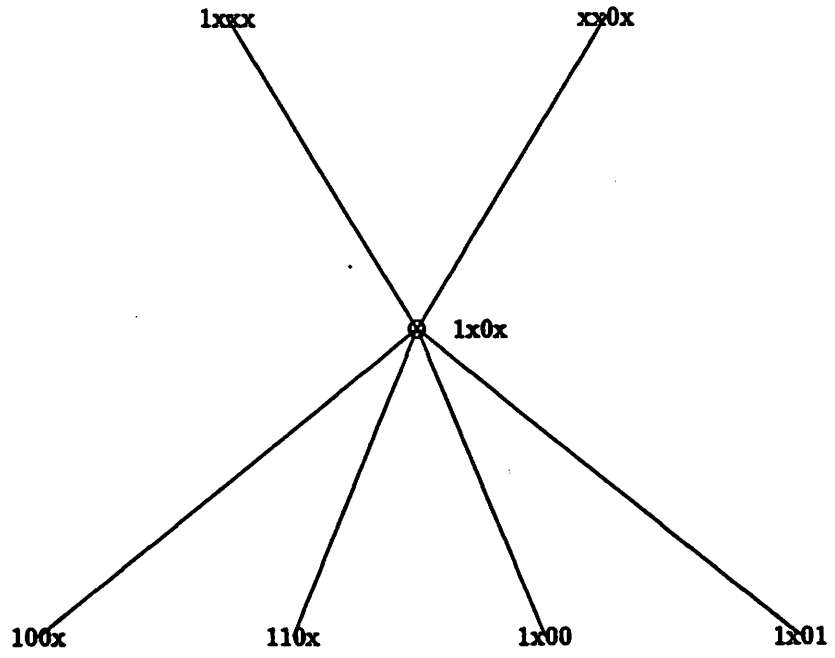


Fig. 3.2. A face with its vertical neighborhood

Assign_face is the routine that maps constraints to faces . Given a constraint, it assigns it a face compatible with the partial assignment built up to now; also, it assigns faces to those constraints of category 2, that are children of the constraint being currently encoded and have another father already encoded . When unable to map the constraint to a face, it informs *backtrack_down* which takes actions, according to the outlined lower backtracking scheme . Faces are generated calling the routine *gen_newface*, and they are verified with help of the routines *fathers_codes_ok* and *code_verify* . We will examine in 3.3.4 the issues brought up by the generation of faces.

We see now the verification aspect . It guarantees the incremental correctness of the assignment . We suppose that up to the i-th step we built a correct partial assignment, i.e. an assignment to a subset of constraints that verifies the subposet equivalence among the constraints already taken into consideration . Coding a new constraint, we want to make sure that we still get a correct assignment with respect to the enlarged set of encoded constraints and of inclusion/intersection relations holding among them . The verification on the instance poset is done by *fathers_code_ok* . The checks are :

1) if the new constraint has only one father, the latter's face must include the face proposed for the former (inclusion condition $S_i \supset S_j$ —> (implies) $f(S_i) \supset f(S_j)$);

2) if the new constraint has more than one father, the faces assigned to the fathers must intersect in the face proposed for the child (intersection condition $S_i \bigcap S_j = S_k$ —> (implies) $f(S_i) \bigcap f(S_j) = f(S_k)$) . We limit the check to the fathers of the constraints being encoded, because we build the global assignment function incrementally fathers first, children after, and so on the instance poset we need to worry only about the local fathers/children relations .

The verification on the face-poset is done by *code_verify* . The checks are :

1) the face proposed for the constraint being encoded must be different from the faces already assigned (the mapping has to be injective);

2a) if an assigned face includes properly the face proposed, the former's inverse must be a father of the constraint being encoded,

2b) if the face proposed includes properly a face already assigned, the latter's inverse has a to be a child of the constraint being encoded (both verify the inclusion condition $f(S_i) \supset f(S_j)$ —> (implies) $S_i \supset S_j$);

3) for all other constraints already coded at the same level, that have a non-void intersection with the one being encoded it must be true that $f(S_i) \bigcap f(S_j) = f(S_k)$ —> (implies) $S_i \bigcap S_j = S_k$ (intersection condition) . On the face-poset the check is global, because a new proposed face may a priori lay anyway in it .

Inductively, we can say that we always guarantee a correct partial assignment, so when we are able to extend it to the complete instance poset, we have a correct solution of the problem . When a complete assignment has been successfully produced by the algorithm, a different verification routine checks again the encoding, going through all the constraints and faces, to warn of bugs in the implementation, if any .

3.3.4. The generation of faces

The routine *gen_newface* walks through the n-face-poset . Given a constraint being currently encoded, *gen_newface* returns a new face, that *assign_face* checks for correctness . If it isn't, *gen_newface* is called again until a face eventually passes the test of correctness or *gen_newface* exhausts all possible faces of the face-poset of proper cardinality for the constraint and that satisfy, when known, some locality conditions . If a constraint is of category 3, the locality condition restricts the potential

The meaning of the first symbolic implicant above is "when input input_1 is asserted , proceed from state state_1 to state state_3 with the first, second, third and fifth outputs low, and the fourth output high". Note that the symbolic implicants are in one-to-one correspondence with the arcs in a state-diagram representation of the FSM.

The following options are understood by *nova*:

.list – The input finite state machine table is printed on the standard output .

.symbolic input – Inputs are considered as symbolic strings and an optimal assignment of binary vectors to each symbolic input is also performed.

.constraint forcing – Forces (when it is possible) columns of the codes of the next states to coincide with columns of proper outputs. Experimental results show this option scarcely effective , so don't expect much by using it.

.constraint pow2constr – Constraints whose cardinality is not a power of two are either kept in the lattice or deleted , according to the free positions available on the hypercube . Other massages of the constraints lattice are carried on , trying to prune unimportant constraints unlikely to be satisfied by a short code . This option is often very effective .

.constraint forcing pow2constr – Both previous options are set active.

.output complement – All the rotations of the codes are tried . Analytically they correspond to all the column-wise complementations , or said in another way , to setting in turn to zero the code of all the states . While it is true that the best choice of the state to code to zero depends on how many times it appears as a next state with a zero proper output - the default heuristic - , sometimes this option finds a better complementation , likely exploiting implicit ordering relations among the outputs .

.random [=integer] – Some random codings are tried . The user can specify how many trials , otherwise the following default values hold : #states if the proper inputs are not symbolic , #states + #inputs if the proper inputs are symbolic . The only purpose of this option is showing you , users , how bad you could do if you don't use carefully chosen assignments , i.e. *nova* !

A sequence of detailed snapshots of the internal running of the program is printed on the standard output . Most are of interest only to the user concerned with the algorithms as implemented internally . The only important information comes at the end under the heading *SUMMARY* .

Example of summary of a run

SUMMARY

```
onehot_products = 24

best_algo = vebucc
best_products = 32
best_size = 640
states[0]:state_1   Best code: 111
states[1]:state_2   Best code: 110
states[2]:state_3   Best code: 011
states[3]:state_4   Best code: 010
states[4]:state_5   Best code: 001
states[5]:state_6   Best code: 101
states[6]:state_7   Best code: 100
states[7]:state_8   Best code: 000

Performance evaluation of the coding on the states constraints
The constraint 00000011 was used and SATISFIED
```

```
The constraint 00110000 wasn't used but luckily SATISFIED
The constraint 10000100 was used and SATISFIED
The constraint 00010001 was used and SATISFIED
The constraint 00000110 was used and SATISFIED
The constraint 00001100 was used and SATISFIED
The constraint 00100010 wasn't used and remained UNSATISFIED
The constraint 00101000 was used and SATISFIED
The constraint 00001001 was used and SATISFIED
The constraint 11000000 was used and SATISFIED
measure(satisfaction) = 9
measure(unsatisfaction) = 1

THE END of NOVA
```

The field *best_size* gives the final minimized PLA area according to this encoding . The user should compare this figure with the ones obtained running nova with different options to find when a minimum area implementation is achieved.

Other figures are :
onehot_products = # of product terms of the 1-hot coded pla
best_products = # of product terms obtained with this encoding
measure(satisfaction) = cumulative weight of the constraints satisfied by this encoding
measure(unsatisfaction) = cumulative weight of the constraints unsatisfied by this encoding

The codes and information on the usage of the constraints are reported also .

A complete session could require running the input with the following constraint-related options :

a) no special option

b) *.constraint forcing*

c) *.constraint pow2constr*

d) *.constraint forcing pow2constr*

e) *.output complement*

f) *.constraint pow2constr*
.output complement

g) *.constraint forcing pow2constr*
.output complement

Sample of a running session with the input file dk14 (notice that the inputs are symbolic , so every run requires the option .symbolic input)

> Run with options of case a)

SUMMARY

```
onehot_products = 24
best_products = 32
```

```
best_size = 640

Inputs:
measure(satisfaction) = 9
measure(unsatisfaction) = 1

States:
measure(satisfaction) = 22
measure(unsatisfaction) = 18
THE END of NOVA
```

> Run with options of case b)

```
                                        SUMMARY
onehot_products = 24
best_products = 35
best_size = 700

Inputs:
measure(satisfaction) = 9
measure(unsatisfaction) = 1

States:
measure(satisfaction) = 28
measure(unsatisfaction) = 23
THE END of NOVA
```

Given the unsatisfactory result obtained with options of case b) we can skip the runs with options of cases d) and g) .

> Run with options of case c)

```
                                        SUMMARY
onehot_products = 24
best_products = 28
best_size = 560

Inputs:
measure(satisfaction) = 9
measure(unsatisfaction) = 1

States:
measure(satisfaction) = 19
measure(unsatisfaction) = 5
THE END of NOVA
```

> Run with options of case e)

```
                                        SUMMARY
onehot_products = 24
best_products = 30
best_size = 600

Inputs:
measure(satisfaction) = 9
measure(unsatisfaction) = 1

States:
measure(satisfaction) = 22
measure(unsatisfaction) = 18
THE END of NOVA
```

> Run with options of case f)

```
                                        SUMMARY
onehot_products = 24
best_products = 27
best_size = 540

Inputs:
measure(satisfaction) = 9
measure(unsatisfaction) = 1

States:
measure(satisfaction) = 19
measure(unsatisfaction) = 5
THE END of NOVA
```

The best result (area=540) is obtained with options of case f).

SEE ALSO

espresso(CAD1)

AUTHOR

Tiziano Villa (villa@ucbic)

COMMENTS

In a given state, if symbolic implicants are not specified for all possible input conditions, then the state machine response for the unspecified conditions is undefined. In particular, *nova* will use this to its advantage when assigning the state codes. It is possible to see all of the don't cares created in this way by using the -out fd option when the PLA is minimized with *espresso*.

It is possible to specify logically inconsistent finite state machines (i.e., to specify two transitions for the same set of inputs in a single state) and this should be, but is not, detected as an error.

Temporary files *temp1* , *temp2* , *temp3* , *temp4* and *temp5* are created in the current working directory . Their meaning is as follows .

temp1 : 1-hot coded cover of the input table . It is fed into the minimizer *Espresso* invoked in the multiple-valued mode to get the input constraints.

temp2 : output of the above specified run of the minimizer *Espresso* . Its cardinality gives the lower bound on the product terms of the coded cover , with respect to the input constraints relations.

temp3 : boolean coded cover of the input table . It is fed into the minimizer *Espresso* invoked in the normal mode.

temp4 : output of the above specified run of the minimizer Espresso .

temp5 : best coded minimized pla implementation . It can be fed (with the proper synctatic adaptations) into a pla layout generator .

The above mentioned files are not removed in the current version and can be inspected by the user to verify an intermediate step of the logical mimizations . Therefore, it is wise to avoid multiple *nova* runs in the same directory at the same time.

Only a single symbolic input (besides the present state) is allowed. The ability to specify any number of symbolic inputs along with binary inputs would be much more practical.

Options not understood by the program are ignored.

nova invokes the multiple-valued minimization program *espresso* (CAD1).

nova is written in C. There are no limitations on the number of binary or symbolic inputs, binary outputs, states, or symbolic implicants.

A message like follows (rarely issued) warns only that the detection of the lattice intersections has been stopped after a quite large number of them has been computed . No action needs to be taken .

Message fac-simile :

WARNING

"After that lattice added the 1001-th new constraint , Nova stopped executing lattice and went ahead with the constraints that lattice already got" .

(implementatio) nova comes from latin and means *a new (implementation).* No connections to astronomical objects is implied.

faces to those included in the face already assigned to the father . No constraints of category 2 are serviced by *gen_newface* , since they are encoded as a byproduct of assigning faces to their fathers . Only constraints of category 1 pose a problem when generating new faces, because potentially some of them require the exploration of all possible faces of a given cardinality . The point is that, when generating most or all faces, we don't want to store those already tried to know which ones have been rejected; instead, we prefer to produce them in an orderly fashion, that makes controlling the generation very quick .

Then, we use the information about inclusion and intersection relations to start from a seed, i.e. a face that is a reasonable starting neighbor for the new face, and generate faces expanding in the free neighborhood . If none of them passes the checks of *assign_face* , we trigger the ordered generation of all possible faces of proper cardinality . Currently we implemented a mechanism based on the generation of all combinations of the x's of the face in lexicographic order, we plan also to implement a Gray reversed code that minimizes the amount of change from one combination to the other and a random criterion to evaluate the heuristics . It is interesting to come up with an orderly generation mechanism that produces the faces according to some decreasing degree of satisfiability, although in general we can't expect that a criterion work for all cases .

3.4. Coding singletons using don't cares

It is an interesting issue in optimal state assignment to investigate the usefulness of assigning don't cares to states . The encoding mechanism implemented in [MSV83] allowed it; the techniques of KISS and NOVA didn't use them . Here we want to analyze whether our way of modeling CONSTRAINED CUBICAL EMBEDDING via SUBPOSET DIMENSION allows or not coding states with don't cares .

In our model, all constraints can be assigned faces of a dimension larger than the minimum required by the cardinality of the constraint . Singleton constraints, i.e. constraints representing single states or symbols, are no exception, so they too can be assigned faces larger then their cardinality (one), which means faces with don't cares . There are cases when this allows to achieve the encoding in a smaller hypercube that otherwise possible . Consider the following example of Figure 3.3 . Not allowing the use of don't care with singletons, it would be necessary a 4-cube to encode the constraints , as shown in Figure 3.4 . Allowing the use of don't care with singletons, we can embed the given constraints in a 3-cube , as shown in Figure 3.5 .

There are other cases when it would be useful to introduce don't cares, but it is not allowed by our model . Consider the power set of {1,2,3} in Figure 3.6 . It can be coded in a 2-cube assigning an edge to the state 2 , as shown in Figure 3.7 . Our algorithm, instead, would find the solution shown in Figure 3.8 . That solution is the correct one in our model; we are dealing with the all-constrained case on 3 states, we need a 3-cube, as the upper bound teaches us .

We tend to see these as "anomalies" due to the difficulty of mapping completely CONSTRAINED CUBICAL EMBEDDING in purely combinatorial terms . We don't think that they are going to penalize our algorithm in any serious sense, although we pointed them out, as limit cases in which the model doesn't capture all the problem .

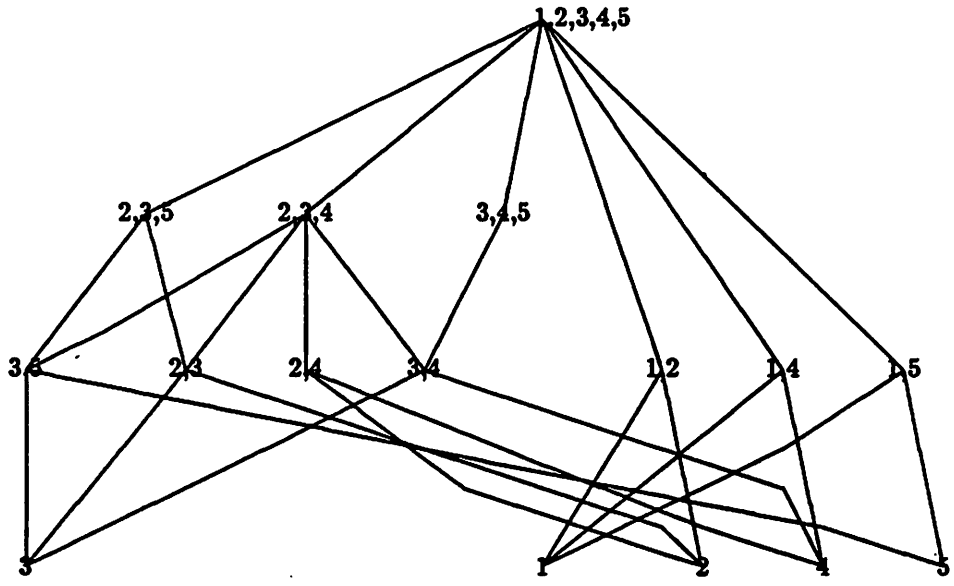


Fig. 3.3. Instance poset

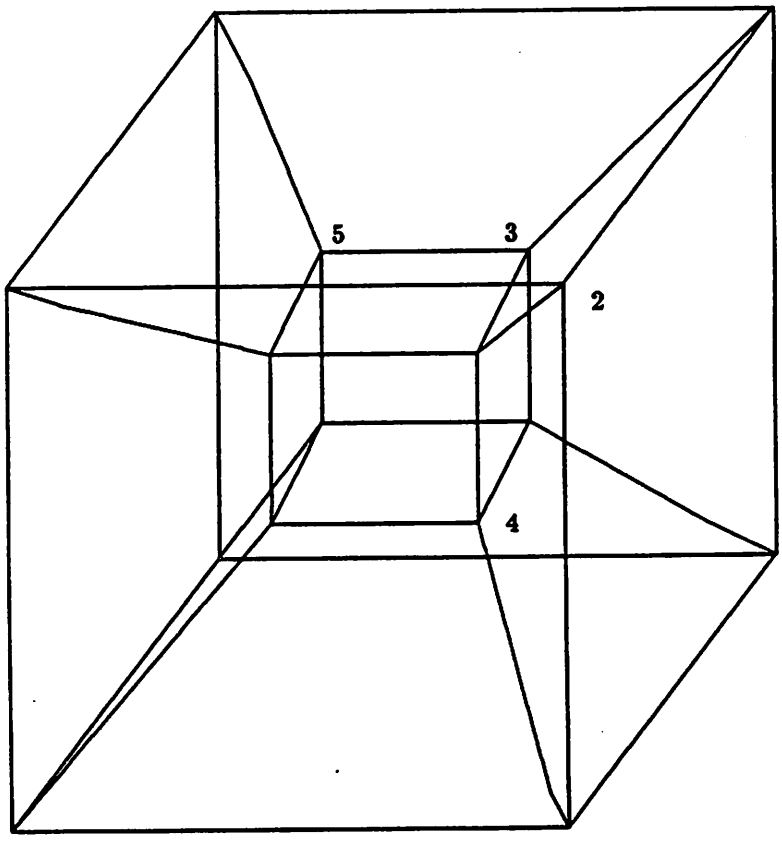


Fig. 3.4. Not using "x" with singletons in ex. of Figure 3.3

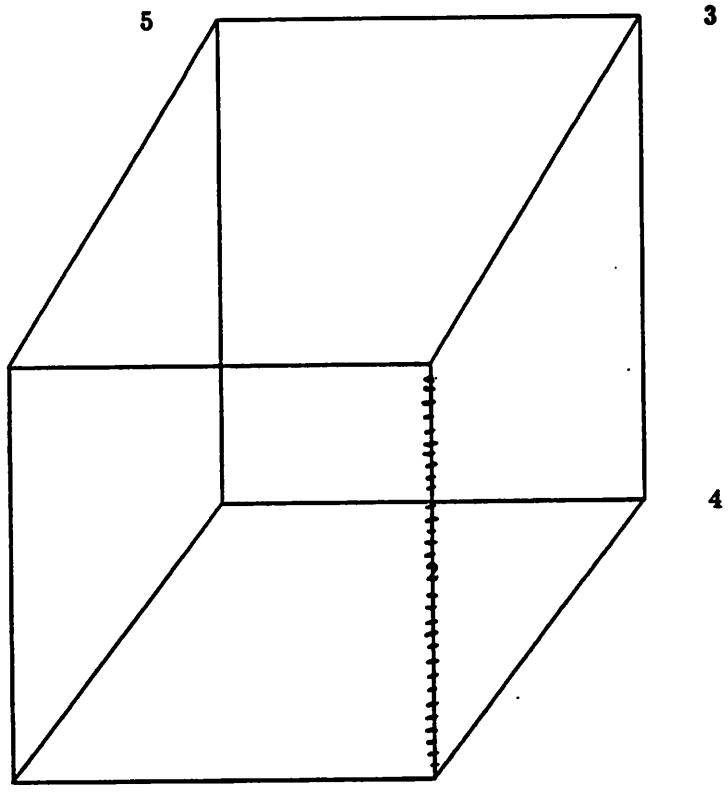


Fig. 3.5. Using "x" with singletons in ex. of Figure 3.3

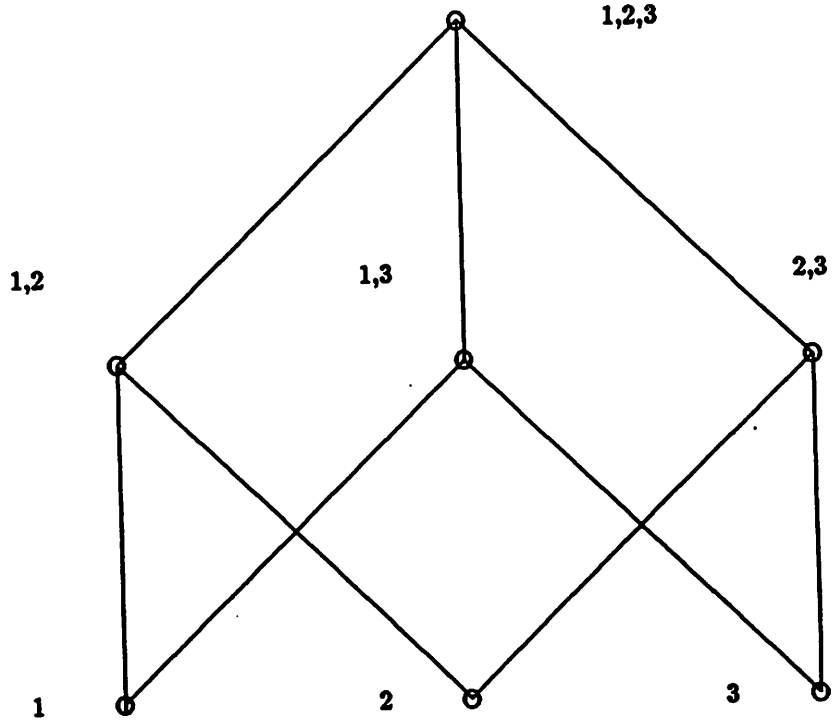


Fig. 3.6. Complete poset on three elements

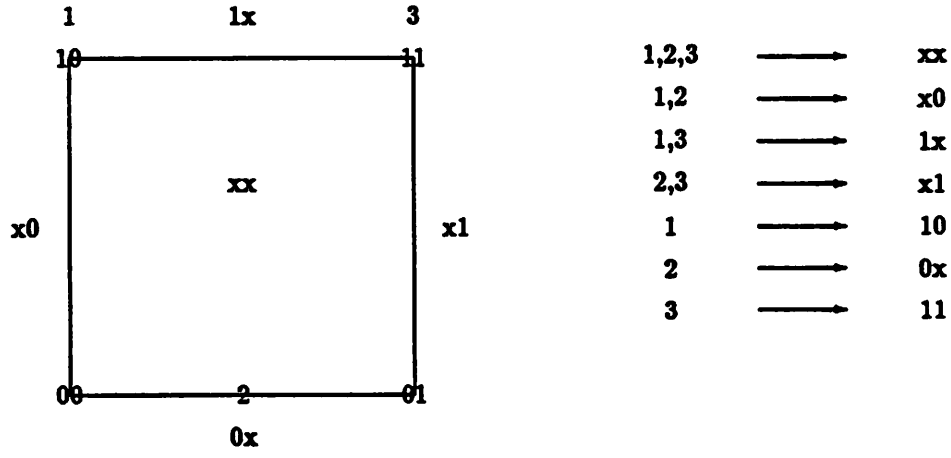
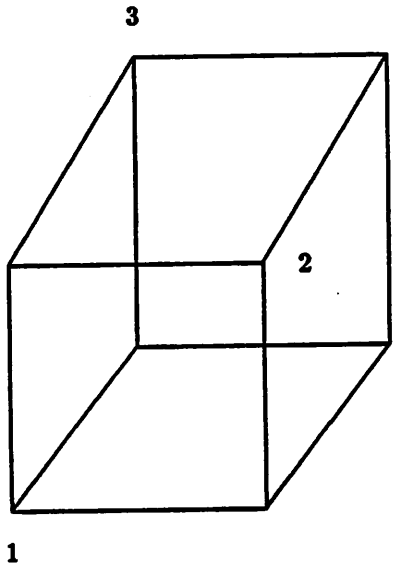


Fig. 3.7. Optimal solution to the ex. of Figure 3.6



1,2,3	⟶	xxx
1,2	⟶	x0x
1,3	⟶	0xx
2,3	⟶	xx1
1	⟶	000
2	⟶	101
3	⟶	011

Fig. 3.8. Our solution to the ex. of Figure 3.6

3.5. An example

In this section, we show an exact solution to a non-trivial instance, requiring a 5-face-poset . Finding the exact solution allowed in this case to achieve a minimum area compared to both NOVA and KISS (see table 4.1) .

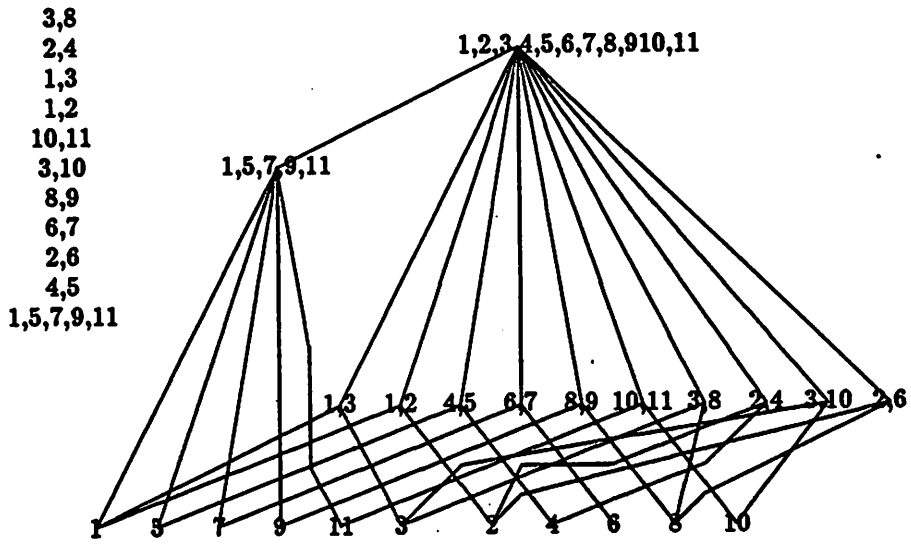


Fig. 3.9. An example : constraints and instance poset

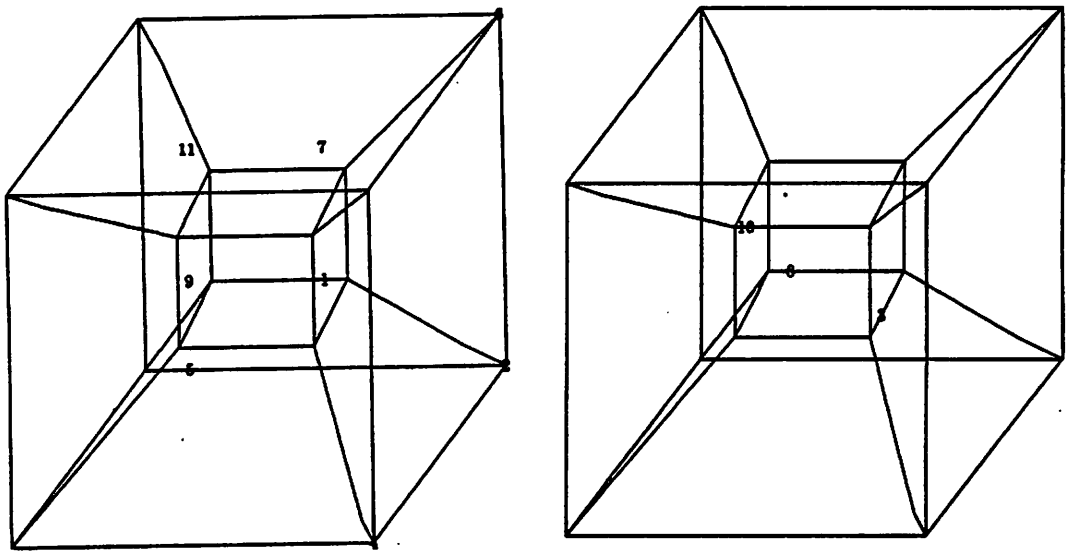


Fig. 3.10. Exact solution of ex. of Figure 3.9

3.6. Experimental results on optimal encoding of FSMs

We have a prototype of the exact algorithm already working (coded in C, approximately 2,500 lines of code), although not all described features are already there . In table 3.1 the results of the exact algorithm applied to the optimal encoding of some FSMs are given . They are checked against the results of the programs KISS and NOVA . (see Chapter 4) . Compared to KISS, the exact algorithm achieves almost always a smaller number of product terms and a shorter code-length . In terms of area, NOVA wins in all reported examples, except one where the exact algorithm gives the best . We need more work to speed up the exact algorithm and much more testing to make

stronger statements . Moreover, it will be important to test the exact algorithm on other problems of constrained embedding besides state assignment of FSMs .

	1-hot	Exact solution			NOVA			KISS		
	(a)	(b)	(c)	(d)	(b)	(c)	(d)	(b)	(c)	(d)
FSM1	24	4+4	22	550	3+3	27	540	4+5	25	700
FSM2	25	4	23	529	3	26	520	4	23	529
FSM3	17	3+3	16	320	3+2	18	306	3+4	17	391
FSM4	17	4	16	368	2	18	306	4	16	368
FSM5	20	2+4	17	323	2+3	18	288	2+4	19	361
FSM6	21	1+5	17	340	1+4	17	289	1+6	18	414
FSM7	77	7	49	1813	5	49	1519	8	47	1880
FSM8	11	5	9	180	4	11	187	6	10	230

(a) #product-terms of minimized multiple-valued cover
(b) [code-length of symbolic inputs +] code-length of states
(c) #product-terms of minimized coded FSM
(d) area of minimized coded FSM

Table 3.1 Experimental results

CHAPTER 4

NOVA: An approximate algorithm to solve constrained embedding

Understanding by experiments that increasing too much the length of the codes of the states of a FSM, it isn't likely to pay in terms of area, together with practical limitations of KISS, (the program in use at UCB for optimal state assignment), motivated us to design a new algorithm and implement the program NOVA . The philosophy of this new encoding algorithm, is never to increase the length of the code to satisfy more constraints, but instead it tries to satisfy heuristically as many constraints as it can with minimum code length (log of cardinality of the state set) . We will analyze the main features of this algorithm in the next paragraphs . Its basic scheme is :

```
{

   /* determines output-related constraints, if any */
   if (FORCING) output_forcing();

   /* processes the constraints */
   precode();

   /* assigns the code */
   coding_cycle();

   /* rotates the code */
   rotation();

   /* tries random codes */
   if (RANDOM) random_test();
```

```
    /* chooses the best code obtained so far */
    choose_best_result();
}
```

4.1. Output forcing

The routine *output_forcing* . supplements the pool of constraints determined by the initial multiple-valued logic minimization with new constraints expressing the condition that some columns carrying next-state information are forced to coincide with columns of proper outputs (i.e. not symbolic), when it is feasible . It is invoked under user's control, by setting true the boolean parameter FORCING . Output_forcing achieves its goal by cycling through all proper output variables; for each of them, it verifies if the next states can be roughly partitioned into two sets of the same size SET_{0_i} and SET_{1_i} (the index i refers to the proper output column):

SET_{0_i} contains all next states with i-th proper output equal to 0,

SET_{1_i} contains all next states with i-th proper output equal to 1 .

When this is possible, a new ("output") constraint expressing the condition that all states in the same subset should be encoded in the same face of the hypercube is added to the collection of constraints . This would allow a likely simplification of the combinational logic and a sure decrease in the column cardinality, because the same column could be shared by both next state and proper output . This feature was suggested by the practice of some logic designers when encoding FSMs . Our experiments didn't find it effective in most cases, nethertheless in a few situations it allowed a dramatic decrease in area .

4.2. Processing the collection of constraints

The routine *precode* processes the collection of constraints . It deletes some of them (those unsatisfiable in the given hypercube) and intersects the others to obtain their closure with respect to set-theoretic intersection (called intersecting family) . The intersecting family is basic to the following coding step . The scheme of precode is :

```
{
  if (POW2CONSTR) {

    /* deletes the pow2-uneven constraints */
    pow2pruning();

    /* computes the intersecting family */
    constraint_closure();

    /* deletes the pow2-uneven constraints */
    pow2pruning();

  } else {

    /* computes the intersecting family */
    constraint_closure();
  }
}
```

The routine *pow2pruning* handles the constraints whose cardinality is not a power of two (call them pow2-uneven constraints), deleting some or all of them . It is invoked under user's control, by setting true the boolean parameter POW2CONSTR . Since we embed into an hypercube of dimension equal to the lower bound of the prob-

lem, in order to satisfy pow2-uneven constraints, it is necessary that the hypercube have more vertices than there are states (in this case we say that the hypercube has holes); if this is not so, it is better to delete the pow2-uneven constraints, otherwise we keep as many of them as there is space on the hypercube, i.e. each pow2-uneven constraint, if satisfied, freezes on the hypercube as many vertices as the difference (minimum power of two of the cardinality of the constraint - cardinality of constraint), since we cannot hope of satisfying more pow2-uneven constraints than we can fit their cumulative amount of frozen space in the hypercube . If the hypercube has holes, we try to keep with higher priority the pow2-uneven constraints of larger cardinality . If the user doesn't set POW2CONSTR true, the algorithm will recognize which constraints may or may not be satisfied in the coding step, by actually trying to satisfy them; pruning some of the unsatisfiable ones before the coding (as *pow2pruning* does), may help it in concentrating on feasible constraints , avoiding *faux pas* , as will become clearer when detailing the selection and coding routines .

The routine *constraint_closure* intersects the constraints and adds to them the intersections to get the complete intersecting family . All constraints are classified according to depth, i.e. initially they have all depth = 0 and then, every time that they are equal to the intersection of two constraints of upper depth, their depth is increased by one . Pictorially, the intersecting family can be seen as an Hasse diagram and the depth is the level of a constraint . In case of a very large initial set of constraints, when the cardinality of the intersecting family passes a threshold (conventionally we set as threshold the number 1001), no more intersections are computed, to avoid potential combinatorial explosion (like dealing with the power set of the states) and to ease the work of the encoding algorithm . Obviously, this could degrade in some case the performance of the algorithm .

If POW2CONSTR is true, the routine *pow2pruning* is invoked again after

constraint_closure , because pow2-uneven constraints may be added by the latter .

4.3. Coding the states

The routine *coding_cycle* encodes the states by clusters starting from seed states computed by selection routines. There are two operations involved in the coding cycle:

1) selection of new states as seed of the coding operations (routines *sel_first*, *sel_next*, *sel_propagate*) and

2) assignment of codes to the selected states and other constraint-related states, i.e. states belonging to the same constraints as the seed (routines *code_first*, *code_next*, *code_propagate*) . The scheme of *coding_cycle is :*

```
{

   /* selects the first state to code */
   sel_state = sel_first();

   /* codes sel_state and constraint-related states */
   code_first(sel_state);

   /* loops while there are uncoded state */

   cycle_in_progress = FALSE;

   if (there are uncoded states) {
      cycle_in_progress = TRUE;
   }

   while ( cycle_in_progress == TRUE ) {

      /* returns a coded state; from its code a new cluster
```

```
      of states can be propagated on the hypercube */
    sel_state = sel_propagate();

    if ( sel_state != empty ) {  /* there is a propagation state */

      /* codes a new cluster of states starting from sel_state */
      code_propagate(sel_state);

    } else {

       /* selects a new state to code (no propagation possible
          from the coded states) */
       sel_state = sel_next();

       /* codes sel_state and constraint-related states */
       code_next(sel_state);

    }

    cycle_in_progress = FALSE;

      if (there are uncoded states ) {
       cycle_in_progress = TRUE;
    }

  }

}
```

4.4. Selection routines

The selection routines choose the next state to code . They are *sel_first*, *sel_next* *and sel_propagate* .

The routines *sel_first* and *sel_next* choose a new state not included in any constraint including a state already coded . Their basic structure is :

```
search the deepest constraint not already satisfied;

if (it is found) {

    give to each state a score =
    (10 + weight) * (number of unsatisfied
    constraints of same depth containing it);
    choose the state with maximum score;

} else { search for the first uncoded state }
```

The routine sel_propagate chooses a coded state, called the seed, as a propagation source for new codes . Its basic structure is :

```
for (all coded states) {
   if (state s_i has a code adjacent to free vertices in the cube) {
      if (s_i belongs to an unused constraint containing uncoded states) {
         computes a score for s_i
         ( = depth * weight of unused constraints containing s_i );
      }
   }
}
choose state with maximum score;
```

4.5. Coding routines

The coding routines are *code_first*, *code_next and code_propagate* . The routine *code_first (code_next)* codes the first state (seed state) and the other constrained-related ones starting from the deepest constraints . The routine *code_propagate* propagates from the seed a cluster of codes of uncoded states constraint-related to it . Their basic structure is (here distance means Hamming-distance) :

```
code_first(seed)
seed : state returned by sel_first

assign to seed a conventional code;
for (all unused constraints c (in decreasing depth) that include seed) {
   assign to the uncoded states in c the free vertex at
   minimum distance from the set of coded constraints of c;
   freezes some holes if c is pow2-uneven;
}

code_next(seed)
seed : state returned by sel_next

assign to seed the free vertex at maximum distance from the vertices already
assigned;
for (all unused constraints c (in decreasing depth) that include seed) {
   assign to the uncoded states in c the free vertex at
   minimum distance from the set of coded constraints of c;
   freezes some holes if c is pow2-uneven;
}

code_propagate(seed)
```

seed : state returned by sel_propagate

for (all unused constraints c (in decreasing depth) that include seed) {

assign to the uncoded states in c the free vertex at

minimum distance from the set of coded constraints of c;

freezes some holes if c is pow2-uneven;

}

4.6. Complementation

After the coding is complete and verified, a rotation is applied to it by determining the state that appears more often as a next-state with a null proper output and then giving it the zero boolean code . Among all possible rotations, this should guarantee the bigger reduction of product-terms, since product-terms with all zero outputs are dropped by the minimizer from the final FSM boolean cover . Actually we found experimentally that, sometimes, there are other rotations which obtain a better minimal final cover . We think that this is related to the heuristic nature of the available minimization algorithms, and maybe to some hidden effects of the next-state encoding . To exploit these cases, the user may set true the option COMPLEMENT, that tries all possible rotations of the given code and keeps the best result (routine *choose_best_result*) . All rotations of a code are obtained by complementing in turn all bits of a column of the code matrix . We recall that state assignment is invariant with respect to rotations , and that a n-cube has n distinct rotations .

4.7. Randomization

Because it is difficult to evaluate the quality of a state assignment compared to the absolute best (very rarely known), i.e. independently of the technique used in

assigning the states, we gave the user the possibility of trying random assignments . This is done by setting true the parameter RANDOM and specifying how many cases must be tried by the program . In case at least one of them returns a better result than the official algorithm, the user is warned of the low performance and the better random coding is kept . This outcome is not unusual when multiple-valued logic minimization returns almost no constraints and our embedding technique degenerates to just a random assignment among others .

4.8. Experimental results

For sake of illustration we report a few snapshots from the running of NOVA on an industrial FSM .

Constraints after multiple-valued minimization

00010100 weight:1 depth:0

01000100 weight:1 depth:0

11010000 weight:1 depth:0

00100010 weight:1 depth:0

00001010 weight:1 depth:0

11000000 weight:1 depth:0

10001000 weight:1 depth:0

00110100 weight:1 depth:0

01001100 weight:1 depth:0

00110010 weight:1 depth:0

01001010 weight:1 depth:0

10010000 weight:1 depth:0

00100100 weight:1 depth:0

10100000 weight:1 depth:0

00110000 weight:7 depth:0

01001000 weight:7 depth:0

00000110 weight:7 depth:0

Constraints after precode

10000000 weight:6 depth:2 PRUNED

00001000 weight:3 depth:2 PRUNED

00100000 weight:6 depth:2 PRUNED

00000010 weight:3 depth:2 PRUNED

01000000 weight:3 depth:2 PRUNED

00010000 weight:3 depth:2 PRUNED

00000100 weight:6 depth:2 PRUNED

00010100 weight:1 depth:0

01000100 weight:1 depth:0

11010000 weight:1 depth:0 PRUNED

00100010 weight:1 depth:0

00001010 weight:1 depth:0

11000000 weight:1 depth:0

10001000 weight:1 depth:0

00110100 weight:1 depth:0 PRUNED

01001100 weight:1 depth:0 PRUNED

00110010 weight:1 depth:0 PRUNED

01001010 weight:1 depth:0 PRUNED

10010000 weight:1 depth:0

00100100 weight:1 depth:0

10100000 weight:1 depth:0

00110000 weight:7 depth:0

01001000 weight:7 depth:0

00000110 weight:7 depth:0

Codes given by coding_cycle

s1 code: 000

s7 code: 010

s2 code: 100

s6 code: 101

s3 code: 011

s8 code: 110

s4 code: 111

s5 code: 001

SUMMARY

product-terms of the multiple-valued minimized cover = 86

product-terms of the minimized boolean cover = 66

area of the minimized boolean cover = 1914

Performance of the coding on the constraints

The constraint 00010100 was used but remained UNSATISFIED

The constraint 01000100 was used and SATISFIED

The constraint 11010000 was pruned and UNSATISFIED

The constraint 00100010 was used but remained UNSATISFIED

The constraint 00001010 wasn't used but luckily SATISFIED

The constraint 11000000 was used and SATISFIED

The constraint 10001000 wasn't used and remained UNSATISFIED

The constraint 00110100 was pruned and UNSATISFIED

The constraint 01001100 was pruned and UNSATISFIED

The constraint 00110010 was pruned and UNSATISFIED

The constraint 01001010 was pruned and UNSATISFIED

The constraint 10010000 wasn't used and remained UNSATISFIED

The constraint 00100100 was used and SATISFIED

The constraint 10100000 was used and SATISFIED

The constraint 00110000 was used and SATISFIED

The constraint 01001000 was used and SATISFIED

The constraint 00000110 was used and SATISFIED

measure(satisfaction) = 26

measure(unsatisfaction) = 9

On the same example KISS obtained :

length of the code = 8 bits (= number of states)

product-terms of the minimized boolean cover = 84

area of the minimized boolean cover = 3696

In the appendixes we provide the results of running NOVA with different options on a large collection of FSM's of both industrial and academic origin . For sake of comparison, the results of KISS are also included . NOVA achieves in most cases a smaller area, although on the average it has more product-terms, drawback taken care by the minimum code-length . It never happens, as sometimes with KISS, that NOVA is unable to complete the coding for unfeasible demands of memory and/or time . We didn't bother to give the timing of NOVA, because it is linear in the number of constraints and states; the only potentially expensive step, i.e. the computation of the intersecting family is checked against a threshold to avoid a combinatorial explosion . The fact that NOVA is so fast (a matter of a few seconds also on relatively large

examples) is due to its extreme greediness; this could become a liability (being trapped in a very poor local minimum) on very complex constrained examples, although it is not the case of many FSM examples, that more often suffer from lack of enough constraints . Notice that, when we talk about the timing of NOVA, we don't take into account the running time of calls to ESPRESSO .

CHAPTER 5

Conclusions and future work

Combinatorial theoretical models of the problem of CONSTRAINED CUBICAL EMBEDDING have been examined . A review of previous techniques of embedding graphs into hypercubes has been presented . A new model in terms of poset embedding has been given and new optimization problems have been defined : SUBPOSET DIMENSION and SUBPOSET EQUIVALENCE (the recognition version of the first one) . The latter has been shown to be NP-complete and the former has been shown to be NP-hard .

An algorithm to solve exactly SUBPOSET DIMENSION has been described . We developed it, beside intrinsic combinatorial interest, because it is a necessary tool to make real progress with approximate solutions to CONSTRAINED CUBICAL EMBEDDING based on heuristics . We implemented a prototype of the algorithm .

An approximate algorithm, NOVA, that tries heuristically to satisfy all it can with minimum code length (log of cardinality of the state set) has been also described . Results of an extensive testing are reported. They show that NOVA outperforms KISS in terms of area efficiency and that it can handle also large examples on which KISS is unable to complete successfully .

A table is reported with the results of the exact encoding algorithm applied to the optimal encoding of some FSMs . Compared to KISS, the exact algorithm achieves almost always a smaller number of product terms and a shorter code-length, but NOVA is almost always winner in terms of area .

Future work includes :

1) We need to complete and test thoroughly the implementation of the exact algorithm . Since its worst case complexity is exponential in the size of instance of the input, we need to characterize its performance on the average .

2) With the help of the exact solution, we want to come up with a good evaluation of the "merit" of each constraint in the exact case (i.e. final area satisfying the complete given poset / final area satisfying the given poset without the constraint to evaluate) to define quantitatively the meaning of satisfying "as many constraints as possible", and gauge/enhance the heuristics *a la* NOVA that limit the dimension of the cube to a fixed value and try to satisfy as many constraints as possible within the given boolean space . We plan to use these findings to improve the current version of NOVA, allowing also for more flexibility in setting the dimension of cube .

3) We plan to derive new heuristics that satisfy all constraints, as suboptimal cases of the exact algorithm . Given that our experimental results for optimal encoding of FSMs show poor gains in area even with an exact embedding when the minimum dimension grows too much with respect to the information-theoretic minimum, algorithms that look for suboptimal satisfaction of all constraints have to be severely tested against exact solutions, to understand what is the best we can hope from them . The approaches in 2) and 3) will be integrated by a master routine, computing the best trade-off between product-terms and code-length to optimize the area .

4) We need to take into account, in the encoding of FSMs, also the order relations among next-state symbols, adjusting the combinatorial analysis to the modified setting .

5) We want to characterize the cases when multiple-valued logic minimization and

next-state ordering don't return enough constraints to trigger a meaningful state assignment . What can then be said of the properties of a good state assignment and how can we use them ?

6) We plan to extend optimal state assignment to multiple-level logic and analyze its interactions with FSM decomposition .

References

References

[APP85] F. Afrati, H. Papadimitriou and G. Papageorgiou, The Complexity of Cubical Graphs, *Inform.Control 66*, (1985), 53-60.

[Arm62] D. B. Armstrong, A Programmed Algorithm for Assigning Internal Codes to Sequential Machines, *IRE Trans.Elect.Comp. EC-11*, (aug. 1962), 466-472.

[BHM84] R. K. Brayton, G. D. Hachtel, C. T. McMullen and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1984.

[DeM83] G. DeMicheli, Computer-Aided Synthesis of PLA-based Systems, *Ph.D.Thesis*, Berkeley, 1983.

[DBS85] G. DeMicheli, R. K. Brayton and A. Sangiovanni-Vincentelli, Optimal State Assignment for Finite State Machines, *IEEE Trans.Computer-Aided Design CAD-4(3)*, (July 1985), 269-284.

[DeM86] G. DeMicheli, Symbolic Design of Combinational and Sequential Logic Circuits Implemented by Two-level Logic Macros, *IEEE Trans.Comp.Aided-Design*, , 1986.

[Djo73] D. Z. DjoKovic, Distance-preserving Subgraphs of Hypercubes, *J.Combinatorial Theory Ser.B 14*, (1973), 263-267.

[GaG73] M. R. Garey and R. L. Graham, On Cubical Graphs, *J.Combin.Theory Ser. A 18*, (1973), 263-267.

[GaJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*, Freeman, 1979.

[GrP71] R. L. Graham and H. O. Pollak, On the Addressing Problem for Loop Switching, *Bell System Tech. J. 50*, (1971), 2495-2519.

[GrP72] R. L. Graham and H. O. Pollak, On Embedding Graphs in Squashed Cubes , *Graph Theory and Applications Lecture Notes in Mathematics 303*, (1972), 99-110, Springer-Verlag .

[HaL72] I. Havel and P. Liebl, Embedding the Dichotomic Tree into the Cube, *Cas.Pest.Mat. 97*, (1972), 201-205.

[HaM72] I. Havel and J. Moravel, B-validation of Graphs, *Czechoslovack Math. J. 22*, (1972), 338-351.

[HeK74] S. H. Hechler and P. C. Kainen, Immersion of Digraphs in Cubes, *Israel J. Math. 18*, (1974), 221-233.

[Joh82] D. S. Johnson, The NP-completness Column : an Ongoing Guide, *Journal of algorithms 3*, (1982), 89-99.

[KNR87] S. Kannan, M. Naor and S. Rudich, Implicit Representation of Graphs, *Unpublished manuscript*, , april 1987.

[MSV83] G. D. Micheli, A. Sangiovanni-Vincentelli and T. Villa, Computer-aided synthesis of PLA-based Finite State Machines, *Int.Conf.on Comp.Aid.Des.*, S.Clara, September 1983.

[Ros78] A. L. Rosenberg, Data Encodings and their Costs, *Acta Inform. 9*, (1978), 273-292.

[Rud86] R. Rudell, Multiple Valued Logic Minimization for PLA Synthesis, *Memorandum M86/65*, Berkeley, June 1986.

[Sau72] G. Saucier, State Assignment of Asynchronous Sequential Machines Using Graph Techniques, *IEEE Trans.Comp. C-21*, (march 1972), 282-288.

APPENDIX N. 1

RESULTS OF THE PERFORMANCE OF NOVA ON SOME EXAMPLES AND COMPARISONS WITH KISS

FSM	#symbols	1h-prods	KISS			NOVA				
			bits	prods	area	bits	prods	area	sat	unsat
dk14	inputs:8	24	4+5	25	700	3+3	32	640	9+22	1+18
	states:7					f =	35	700	9+28	1+23
						p2 =	28	560	9+19	1+5
						fp2 =	31	620	9+14	1+13
						compl =	30	600	9+22	1+18
						fcompl =	30	600	9+28	1+23
						p2compl =	26	520	9+19	1+5
						fp2compl =	31	620	9+14	1+13
						Random trials:15		area-aver:809		
dk14x	states:7	25	bits	prods	area	bits	prods	area	sat	unsat
			4	23	529	3	27	540	24	7
						f =	38	760	12	25
						p2 =	28	560	17	6
						fp2 =	37	740	10	16
						compl =	26	520	24	7
						fcompl =	38	760	12	25
						p2compl =	26	520	17	6
						fp2compl =	35	700	10	16
						Random trials:7		area-aver:760		
						best random : =	35	700	13	18
dk15	inputs:8	17	bits	prods	area	bits	prods	area	sat	unsat
	states:4		3+4	17	391	3+2	18	306	1+11	0+2
						f = p2 = fp2 = compl = fcompl = p2compl = fp2compl =				
						Random trials:12		area-aver:376		
dk15x	states:4	17	bits	prods	area	bits	prods	area	sat	unsat
			4	16	368	2	18	306	10	2
						f =				

[Tro75] W. T. Trotter. Embedding Finite Posets in Cubes. *Discrete mathematics 12.* (1975). 165-172.

[Vil86a] T. Villa. NOVA. *EE290H/CS292H CAD Software Manual.* Berkeley. May 1986.

[Vil86b] T. Villa. Experiments on Symbolic Minimization. *EE290H Project Report.* Berkeley. December 1986.

[Vil86c] T. Villa. An Introduction to NOVA : Optimal Assignment of FSMs. *EE290H/CS292H Software Documentation.* Berkeley. May 1986.

[Yao78] A. C. Yao. On the Loop Switching Addressing Problem. *SIAM J.Computing 7,4.* (nov. 1978). 515-523.

				fp2
				compl = 18 306 10 2 fcompl
				p2compl = 19 323 8 1 fp2compl
				Random trials:4 area-aver:378
dk16	inputs:4 states:27	55	bits prods area 2+10 55 2035	bits prods area sat unsat 2+5 74 1628 2+23 0+38 f
				p2 = 76 1672 2+13 0+27 fp2
				compl = 68 1496 2+23 0+38 fcompl
				p2compl = 72 1584 2+13 0+27 fp2compl
				Random trials:31 area-aver:1994
dk16x	states:27	55	bits prods area (out of memory)	bits prods area sat unsat 5 74 1628 22 27 f
				p2 = 73 1606 19 20 fp2
				compl = 71 1562 22 27 fcompl
				p2compl = 71 1562 19 20 fp2compl
				Random trials:27 area-aver:1983
dk17	inputs:4 states:8	20	bits prods area 2+4 19 361	bits prods area sat unsat 2+3 19 304 1+8 0+2 f
				p2 = 19 304 1+8 0+1 fp2
				compl = 18 288 1+8 0+2 fcompl
				p2compl = 18 288 1+9 0+1 fp2compl
				Random trials:12 area-aver:368

dk27	inputs:2 states:7	10	bits 1+3	prods 9	area 117	bits 1+3	prods 8	area 104	sat 4	unsat 1
						f				
						p2				
						=	7	91	5	0
						fp2				
						compl				
						=	7	91	4	1
						fcompl				
						p2compl				
						=	7	91	5	0
						fp2compl				
						Random trials:9		area-aver:140		
dk512	inputs:2 states:15	21	bits 1+6	prods 18	area 414	bits 1+4	prods 19	area 323	sat 0+7	unsat 0+4
						f				
						p2				
						=	17	289	0+8	0+2
						fp2				
						compl				
						=	19	323	0+7	0+4
						fcompl				
						p2compl				
						=	17	289	0+8	0+2
						fp2compl				
						Random trials:17		area-aver:418		
iofsm	states:10	19	bits 4	prods 16	area 448	bits 4	prods 16	area 448	sat 2	unsat 0
						f				
						p2				
						=	16	448	2	0
						fp2				
						compl				
						=	16	448	2	0
						fcompl				
						p2compl				
						=	16	448	2	0
						fp2compl				
						Random trials:10		area-aver:579		
mark1	states:15	19	bits 4	prods 19	area 722	bits 4	prods 21	area 798	sat 3	unsat 2
						f				
						p2				
						=	19	722	5	0
						fp2				
						compl				
						=	17	646	3	2
						fcompl				
						p2compl				

```
                                                  ==========
                                                  Random
                                                  trials:15  area-aver:782
                                                  best random :
                                                  =    19    722   1     4

mc      states:4     10      bits prods area   bits prods area  sat   unsat
                             2    9     153    2    9     153   0     0
                                                  f
                                                  ==========
                                                  p2
                                                  =    9     153   0     0
                                                  fp2
                                                  ==========
                                                  compl
                                                  =    9     153   0     0
                                                  fcompl
                                                  ==========
                                                  p2compl
                                                  =    9     153   0     0
                                                  fp2compl
                                                  ==========
                                                  Random
                                                  trials:4   area-aver:157

mfsm    states:32    ?       bits prods area   bits prods area  sat   unsat
                             (ON-SET and OFF-SET are not orthogonal)
                             ?    ?     ?      5    60    3060  ?     ?
                                                  f
                                                  ==========
                                                  p2
                                                  ==========
                                                  fp2
                                                  ==========
                                                  compl
                                                  =    55    2805  0     0
                                                  fcompl
                                                  ==========
                                                  p2compl
                                                  ==========
                                                  fp2compl
                                                  ==========
                                                  Random
                                                  ==========

scf     states:121   154     bits prods area   bits prods area  sat   unsat
                             8    140   18760  7    145   18995 13    7
                                                  f
                                                  ==========
                                                  p2
                                                  =    143   18733 14    6
                                                  fp2
                                                  ==========
                                                  compl
                                                  =    145   18995 13    7
                                                  fcompl
                                                  ==========
                                                  p2compl
                                                  =    143   18733 14    6
                                                  fp2compl
                                                  ==========
                                                  Random
                                                  ==========

bbara   states:10    34      bits prods area   bits prods area  sat   unsat
                             5    26    650    4    25    550   5     5
                                                  f
                                                  ==========
```

```
                                                fp2
                                                ==========
                                                compl
                                                =    25    550   5     5
                                                fcompl
                                                ==========
                                                p2compl
                                                =    25    550   7     1
                                                fp2compl
                                                ==========
                                                Random
                                                trials:10  area-aver:649

bbsse    states:16    29        bits prods area  bits prods area  sat   unsat
                                6    27    1053  4    31    1023  6     1
                                                f
                                                ==========
                                                p2
                                                =    31    1023  2     1
                                                fp2
                                                ==========
                                                compl
                                                =    31    1023  6     1
                                                fcompl
                                                ==========
                                                p2compl
                                                =    31    1023  2     1
                                                fp2compl
                                                ==========
                                                Random
                                                trials:16  area-aver:1144

bbtas    states:5     16        bits prods area  bits prods area  sat   unsat
                                3    13    195   3    12    180   0     1
                                                f
                                                ==========
                                                p2
                                                =    13    195   1     0
                                                fp2
                                                ==========
                                                compl
                                                =    11    165   0     1
                                                fcompl
                                                ==========
                                                p2compl
                                                =    11    165   1     0
                                                fp2compl
                                                ==========
                                                Random
                                                trials:5   area-aver:215

beecount states:7     12        bits prods area  bits prods area  sat   unsat
                                4    11    242   3    12    228   7     3
                                                f
                                                ==========
                                                p2
                                                =    12    228   3     3
                                                fp2
                                                ==========
                                                compl
                                                =    10    190   7     3
                                                fcompl
                                                ==========
                                                p2compl
                                                =    11    209   3     3
                                                fp2compl
                                                ==========
                                                Random
                                                trials:7   area-aver:293
```

cse	states:16	55	bits	prods	area	bits	prods	area	sat	unsat
			5	47	1692	4	48	1584	14	6
						f				
						p2				
						=	46	1518	13	0
						fp2				
						compl				
						=	48	1584	14	6
						fcompl				
						p2compl				
						=	46	1518	13	0
						fp2compl				
						Random				
						trials:16		area-aver:2087		
donfile	states:24	24	bits	prods	area	bits	prods	area	sat	unsat
			12	24	984	5	48	960	21	39
						f				
						p2				
						=	48	960	21	39
						fp2				
						compl				
						=	41	820	21	39
						fcompl				
						p2compl				
						=	41	820	21	39
						fp2compl				
						Random				
						trials:24		area-aver:1360		
keyb	states:19	77	bits	prods	area	bits	prods	area	sat	unsat
			8	47	1880	5	55	1705	55	113
						f				
						p2				
						=	49	1519	87	18
						fp2				
						compl				
						=	55	1705	55	113
						fcompl				
						p2compl				
						=	49	1519	87	18
						fp2compl				
						Random				
						trials:19		area-aver:3416		
lion	states:4	8	bits	prods	area	bits	prods	area	sat	unsat
			2	6	66	2	6	66	4	0
						f				
						p2				
						=	6	66	4	0
						fp2				
						compl				
						=	6	66	4	0
						fcompl				
						p2compl				

				Random trials:4 area-aver:96	
lion9	states:9	10	bits prods area 4 8 136	bits prods area 4 9 153	sat 11 unsat 9
				f = p2 = fp2	
				8 136	13 7
				compl = fcompl	
				9 153	11 9
				p2compl = fp2compl	
				8 136	13 7
				Random trials:9 area-aver:266	
modulo12	states:12	24	bits prods area 4 14 210	bits prods area 4 12 180	sat 0 unsat 0
				f = p2 = fp2	
				12 180	0 0
				compl = fcompl	
				11 165	0 0
				p2compl = fp2compl	
				11 165	0 0
				Random trials:12 area-aver:192	
planet	states:48	92	bits prods area 6 89 4539	bits prods area 6 91 4641	sat 10 unsat 2
				f = p2 = fp2	
				88 4488	11 1
				compl = fcompl	
				87 4437	10 2
				p2compl = fp2compl	
				87 4437	11 1
				Random trials:12 area-aver:192	
s1	states:20	92	bits prods area 5 81 2997	bits prods area 5 83 3071	sat 14 unsat 1
				f = p2 = fp2	
				81 2997	15 0

						=	83	3071	14	1	
						fcompl					
						p2compl					
						=	81	2997	15	0	
						fp2compl					
						Random					
						trials:?		area-aver:?			
sand	states:32	114	bits	prods	area	bits	prods	area	sat	unsat	
			6	96	4704	5	102	4692	9	1	
						f					
						p2					
						=	101	4646	8	1	
						fp2					
						compl					
						=	99	4554	9	1	
						fcompl					
						p2compl					
						=	100	4600	8	1	
						fp2compl					
						Random					
						trials:?		area-aver:?			
shiftreg	states:8	9	bits	prods	area	bits	prods	area	sat	unsat	
			3	6	72	3	11	132	6	3	
						f					
						p2					
						=	11	132	6	3	
						fp2					
						compl					
						=	11	132	6	3	
						fcompl					
						p2compl					
						=	11	132	6	3	
						fp2compl					
						Random					
						trials:100		area-aver:132			
styr	states:30	111	bits	prods	area	bits	prods	area	sat	unsat	
			?	?	?	5	101	4343	27	20	
						f					
						p2					
						=	92	3956	22	2	
						fp2					
						compl					
						=	101	4343	27	20	
						fcompl					
						p2compl					
						=	92	3956	22	2	
						fp2compl					
						Random					
						trials:?		area-aver:?			
tav	states:4	12	bits	prods	area	bits	prods	area	sat	unsat	

```
                                                    ==========
                                                    p2
                                                    =    11    198   0     0
                                                    fp2
                                                    ==========
                                                    compl
                                                    =    11    198   0     0
                                                    fcompl
                                                    ==========
                                                    p2compl
                                                    =    11    198   0     0
                                                    fp2compl
                                                    ==========
                                                    Random
                                                    trials:100 area-aver:198

tbk      states:32    173      bits prods area     bits prods area  sat   unsat
                               ?    ?     ?        5    178   5340  118   2507
                                                    (warning: too many constraints)
                                                    f
                                                    ==========
                                                    p2
                                                    =    173   5190  40    64
                                                    fp2
                                                    ==========
                                                    compl
                                                    ==========
                                                    fcompl
                                                    ==========
                                                    p2compl
                                                    =    173   5190  40    64
                                                    fp2compl
                                                    ==========
                                                    Random
                                                    trials:?   area-aver:?

train11  states:11    11       bits prods area     bits prods area  sat   unsat
                               6    10    230      4    12    204   8     3
                                                    f
                                                    ==========
                                                    p2
                                                    =    11    187   6     5
                                                    fp2
                                                    ==========
                                                    compl
                                                    =    11    187   8     3
                                                    fcompl
                                                    ==========
                                                    p2compl
                                                    =    11    187   6     5
                                                    fp2compl
                                                    ==========
                                                    Random
                                                    trials:11  area-aver:241
```

RESULTS OF THE PERFORMANCE OF KISS AND NOVA ON THE FSMs OF THE SBC

			KISS			NOVA				
FSM	#symbols	1h-prods	bits	prods	area	bits	prods	area	sat	unsat
master	states:15	79	4	72	6408	4	74	6586	3	0
physrec	states:10	38	5	34	1564	4	34	1462	10	1
slave	states:10	46	4	35	2555	4	35	2555	8	0
virmach	states:4	16	2	14	532	2	14	532	0	0
wdcnt	states:9	19	4	16	352	4	16	352	2	0

NAME

nova – State Assignment Program for PLA-based Finite-State Machines

SYNOPSIS

nova *file*

DESCRIPTION

nova is a program that performs an optimal assignment of binary codes to the states of a Finite State Machine (FSM). It does the same job as the program *kiss* and they are perfectly compatible , i.e. the same input file can be given to both programs without any change . If you are already familiar with *kiss* you will find easier the following pages . The state coding generated by *nova* minimizes the number of product-terms required by a PLA implementation of the machine, subject to the condition that the shortest code is used. *kiss* , on the other side, increases the code length as much as needed to satisfy the lower bound on the product term cardinality obtained by minimizing the one-hot coded cover . On the average you can expect shorter codes and more product terms with *nova* , longer codes and fewer product terms with *kiss* ; experimental results show that as for the overall area *nova* does a better job than *kiss* . Anyway, you are advised to run both programs on your specific inputs , keeping the best results (sometimes you could also favor one pla ratio over the other , besides being concerned with the area) . A more capable program (*supernova*?) driven by the area cost function is currently under development . It should be able to trade-off between product terms cardinality and code length to find a minimum of the cost function .

The FSM is described by a symbolic cover which is read from standard input. A symbolic cover is a set of symbolic implicants consisting of four fields corresponding to the FSM inputs, present-states, next-states and outputs respectively. The fields are separated by either blanks or tabs, and all four fields must fit on a single line. To allow comments within the input file, any characters after a pound sign ('#') are ignored.

The FSM states are represented by strings of characters (at most 30 characters). Either the present-state or the next-state may be given as ANY to indicate that the state is a *don't care*. (This is useful, for example, in describing the reset logic for the FSM.)

The inputs to the FSM are represented by a string of characters of **0**, **1**, and **–** (where **–** indicates the symbolic implicant does not depend on the corresponding input). The inputs may also be treated as symbolic inputs (analogous to the way that the present-state is a symbolic input), and *nova* will determine an optimal assignment for the inputs as well (see below).

The outputs from the FSM are also given as a string of characters from the set **0**, **1**, and **–**. A **0** or a **1** indicates that the output must be either low or high (respectively) for this transition. A **–** indicates that, for this transition, the output may be either low or high.

Sample Input File

```
.list
# dk14
input_1 state_1 state_3 00010
input_1 state_2 state_1 01001
input_1 state_3 state_3 10010
input_1 state_4 state_3 00010
input_1 state_5 state_1 01001
input_1 state_6 state_1 01001
input_1 state_7 state_3 10010

input_2 state_2 state_2 01001
input_2 state_5 state_2 01001
input_2 state_6 state_2 01001
input_2 state_1 state_4 00010
```

```
input_2 state_3 state_4 10010
input_2 state_4 state_4 00010
input_2 state_7 state_4 10010

input_3 state_5 state_1 10001
input_3 state_6 state_1 10001
input_3 state_7 state_1 10001
input_3 state_1 state_3 01010
input_3 state_2 state_3 00100
input_3 state_3 state_3 01010
input_3 state_4 state_3 00100

input_4 state_5 state_1 10101
input_4 state_6 state_1 10101
input_4 state_7 state_1 10101
input_4 state_1 state_4 01010
input_4 state_3 state_4 01010
input_4 state_2 state_5 00100
input_4 state_4 state_5 00100

input_5 state_2 state_2 00101
input_5 state_5 state_2 00101
input_5 state_1 state_3 01000
input_5 state_3 state_3 01000
input_5 state_4 state_3 10100
input_5 state_6 state_3 10100
input_5 state_7 state_3 10100

input_6 state_2 state_1 00101
input_6 state_5 state_1 00101
input_6 state_1 state_5 00010
input_6 state_3 state_5 10010
input_6 state_4 state_5 00010
input_6 state_6 state_5 10100
input_6 state_7 state_5 10010

input_7 state_2 state_1 00001
input_7 state_5 state_2 10001
input_7 state_6 state_2 10001
input_7 state_7 state_2 10001
input_7 state_1 state_5 01010
input_7 state_3 state_5 01010
input_7 state_4 state_5 10100

input_8 state_2 state_2 00001
input_8 state_5 state_2 10101
input_8 state_6 state_2 10101
input_8 state_7 state_2 10101
input_8 state_1 state_6 01000
input_8 state_3 state_6 01000
input_8 state_4 state_7 10000
.end
```