Copyright © 1997, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

INPUT ENCODING FOR MINIMUM BDD SIZE: THEORY AND EXPERIMENTS

by

Wilsin Gosti, Tiziano Villa, Alex Saldanha, and
Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M97/22

11 April 1997

INPUT ENCODING FOR MINIMUM BDD SIZE: THEORY AND EXPERIMENTS

by

Wilsin Gosti, Tiziano Villa, Alex Saldanha, and
Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M97/22

11 April 1997

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Input Encoding for Minimum BDD Size: Theory and Experiments

Wilsin Gosti[1] Tiziano Villa[2] Alex Saldanha[3] Alberto Sangiovanni-Vincentelli[1]

[1] Dept. of EECS, University of California, Berkeley, CA 94720
[2] PARADES, Via di S. Pantaleo, 66, 00186 Roma, Italy
[3] Cadence Berkeley Labs, 1919 Addison St., Berkeley, CA 94704-1144

Abstract

In this paper, we address the problem of encoding the state variables of a finite state machine such that the BDD representing the next state function and the output function has the minimum number of nodes. We present a simulated annealing algorithm which finds good solutions. We can claim that it finds good solutions because we also developed an exact algorithm that solves the input encoding problem. The results that our simulated annealing produces are close to the optimum ones that our exact algorithm produces. We provide results of these two algorithms on the MCNC sequential circuits.

1 Introduction

Reduced Ordered Binary Decision Diagrams (ROBDDs or simply BDDs) are a data structure used to efficiently represent and manipulate logic functions. They were introduced by Bryant [Bry86] in 1986. Since then, they played a major role in many areas of Computer Aided Design, including logic synthesis, simulation, and formal verification.

The size of a BDD representing a logic function depends on the ordering of its variables. For some functions, the BDD sizes are linear in the number of variables for one ordering; while they are exponential for the other [Bry92]. Many heuristics have been proposed to find good orderings, e.g., the sifting dynamic reordering algorithm [Rud93].

BDDs can also be used to represent the characteristic functions of the transition relations of finite state machines. In this case, the size of the BDDs depends not only on variable ordering, but also on state encoding. Meinel and Theobald studied the effect of state encoding on autonomous counters in [MT96b]. They analyzed 3 different encodings: the standard minimum-length encoding, which gives the lower bound of $5n - 3$ internal nodes for an n-bit autonomous counter, the Gray encoding, which gives the lower bound of $10n - 11$ internal nodes, and a worst-case encoding, which gives an exponential number of nodes in n.

The problem of reducing by state encoding the BDD size of an FSM representation is motivated by applications in logic synthesis and verification. Regarding synthesis, BDDs can be used as a starting point for logic optimization. An example is their use in Timed Shannon Circuits [LMSSV95], where the circuits derived are reported to be competitive in area and often significantly better in power. One would like to derive the smallest BDD with the hope that it leads to a smaller circuit derived from it. Regarding verification, re-encoding has been applied successfully to ease the comparison of "similar" sequential circuits [CQC95].

In this paper, we look into the problem of finding the optimum state encoding that minimizes the BDD that represents a finite state machine. We call this problem the *BDD encoding problem.* To the best of our knowledge, this problem has never been addressed before. The work that is related to this paper is from Meinel and Theobald. In the effort to find a good re-encoding of the state variables to reduce the number of BDD nodes, Meinel and Theobald proposed in [MT96a] a dynamic re-encoding algorithm based on XOR-transformations. Although a little slower than the sifting algorithm, their technique was able to reduce the number of nodes in BDDs even in cases when the sifting algorithm could not.

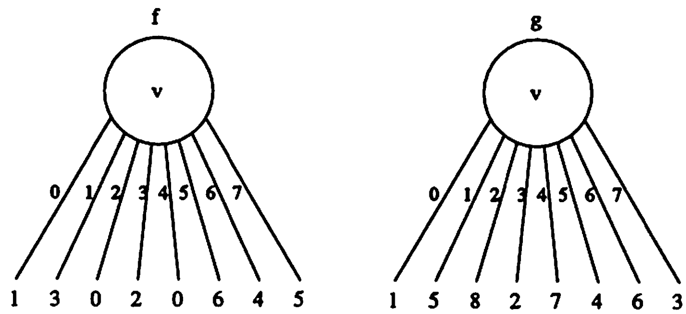


Figure 1: Multi-Valued Functions f and g

As an example, we consider functions (nodes) f and g shown in Figure 1. Functions f and g map $\{0, 1, \ldots, 7\}$ to $\{0, 1, \ldots, 7\}$, and can be regarded as next state functions where the present state variable is v and the next state values are the range. If we encode v as

$$\begin{aligned} e_1(0) &= 010, & e_1(1) &= 100, \\ e_1(2) &= 000, & e_1(3) &= 001, \\ e_1(4) &= 011, & e_1(5) &= 110, \\ e_1(6) &= 111, & e_1(7) &= 101, \end{aligned}$$

with ordering b_2, b_1, b_0 we get the BDD (i.e., part of a BDD) shown in Figure 2 with 14 nodes. *No* reordering will reduce the number of BDD nodes for this encoding.

But if we encode v as

$$\begin{aligned} e_2(0) &= 010, & e_2(1) &= 100, \\ e_2(2) &= 000, & e_2(3) &= 011, \\ e_2(4) &= 001, & e_2(5) &= 111, \\ e_2(6) &= 101, & e_2(7) &= 110 \end{aligned}$$

the BDD that we get has 10 nodes. Figure 3 shows the binary decision trees representing f and g using this encoding. The BDD is shown in Figure 4. From this example, we see that state encodings affect the BDD size representing the transition relation of an FSM.

The remainder of this paper is structured as follows. In Section 2 we state the definitions of FSMs and their BDD representation. We also review briefly the simulated annealing algorithm. In Section 3, we present our simulated annealing algorithm to find an optimal encoding and our experimental results. To evaluate our simulated annealing algorithm, we present an exact formulation of the BDD input encoding problem and our experimental results in Section 4. Finally, we conclude in Section 6.

2 Definitions and Terminology

We review briefly finite state machines and BDDs. We also describe the outline of the simulated annealing algorithm.

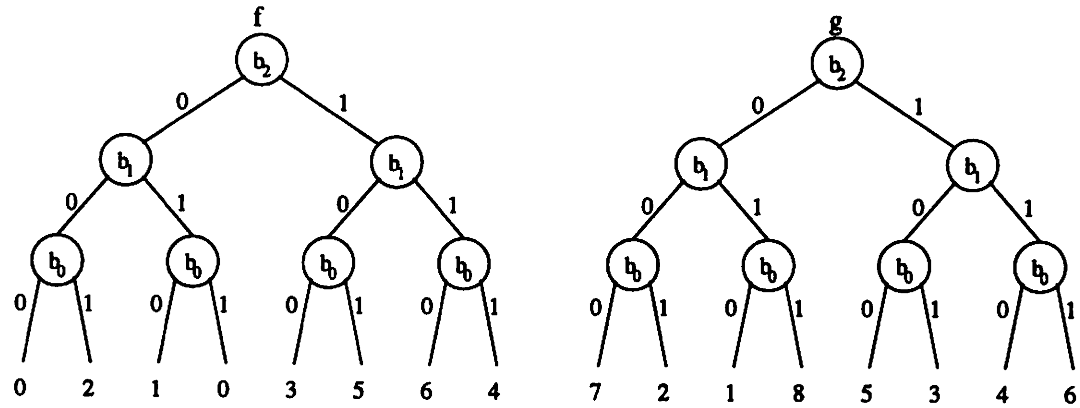


Figure 2: BDD for f and g Using Encoding e_1

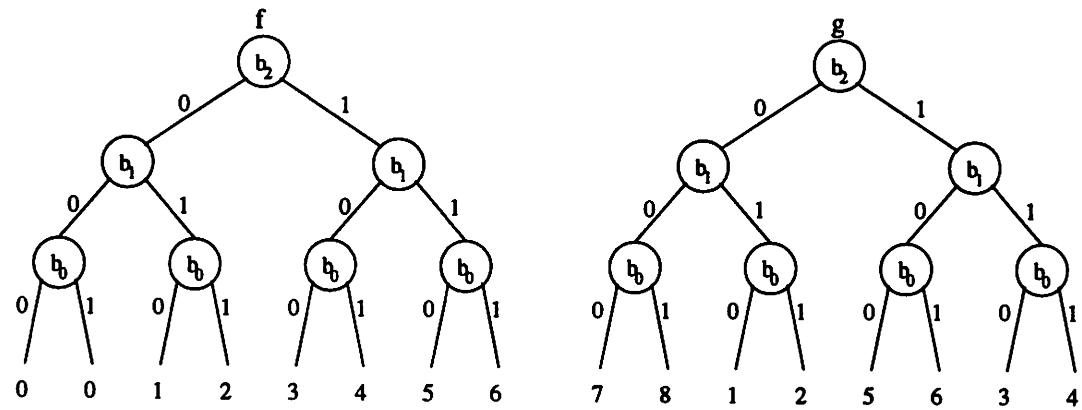


Figure 3: Binary Decision Tree for f and g Using Encoding e_2

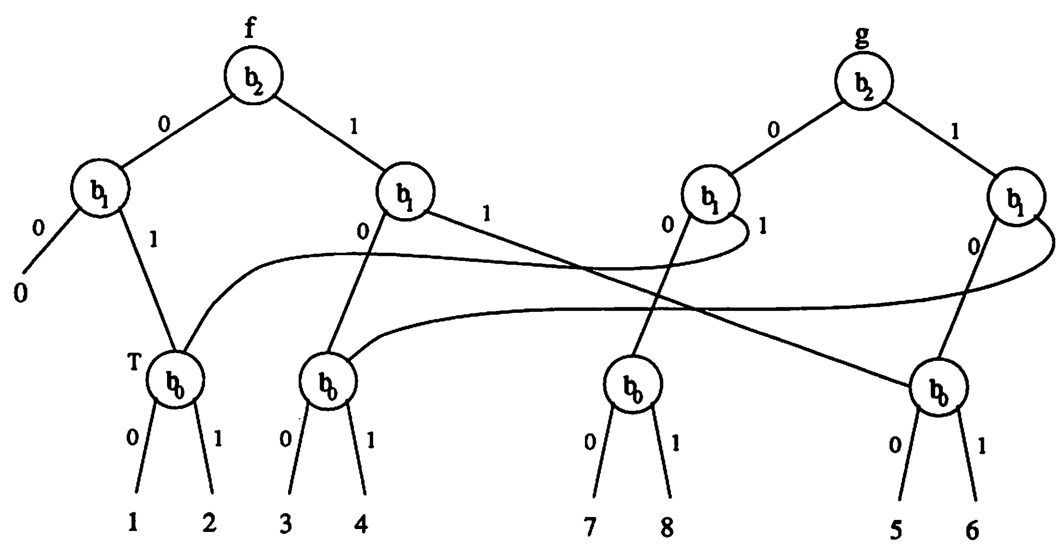


Figure 4: BDD for f and g Using Encoding e_2

2.1 FSMs and Their BDD Representations

Definition 1 *A finite state machine (FSM) is a quintuple $(Q, I, O, \delta, \lambda)$ where Q is the finite set of states, I is the finite set of input values, O is the finite set of output values, δ is the next state function defined as $\delta : I \times Q \mapsto Q$, and λ is the output function defined as $\lambda : I \times Q \mapsto O$.*

An FSM is said to be incompletely specified (ISFSM) if for some input value and present state combination (i, p), the next state or the output is not specified; otherwise, it is said to be completely specified (CSFSM). For an ISFSM, if the next state of (i, p) is not specified, then any state can become the next state. If the output of (i, p) is not specified, then any output can become the output. In the sequel, we will deal only with CSFSMs. ISFSMs will be converted to CSFSMs by selecting a next state and/or an output value for an unspecified (i, p). We assume that the inputs and outputs are given in binary forms, and the state variables are given in symbolic forms, i.e., multi-valued.

The next state and output functions of an FSM can be simultaneously represented by a characteristic function $T : I \times Q \times Q \times O \mapsto B$, where each combination of an input value and a present state is related to a combination of a next state and an output value.

Assume that we have an encoding of the states that uses s bits. Let $p_{s-1}, p_{s-2}, \ldots, p_0$ and $n_{s-1}, n_{s-2}, \ldots, n_0$ be the present and next state binary variables. Then for next state n_i, there is a next state function $n_i = \delta_i(p_{s-1}, p_{s-2}, \ldots, p_0)$. Next state and output functions can be represented using BDDs. We call this representation the *functional representation*. We can also represent an FSM using BDDs by representing its characteristic function T. We call it the *relational representation.*

2.2 Simulated Annealing

Simulated Annealing is a combinatorial optimization technique that mimics the physical crystal annealing process that slowly lowers the temperature until it reaches $0°C$ where the crystal is at its global minimum state. The algorithm can be outlined as follows:

```
sa()
  T = starting temperature
  move = initialMove(T)
  while (T != 0) {
    while (stoppingCriterion(T, move) is not met) {
      newMove = nextMove(T, move)
      move = accept(move, newMove)
    }
    T = reduceTemp(T)
  }
```

An implementation of simulated annealing will have to define the starting temperature, the next move function, the stopping criterion, and the temperature lowering function. This is usually called the cooling schedule. The acceptance criterion depends on the cost function that is defined for an implementation. If the new move has a lower cost, then it is always accepted. If it has a higher cost, then it is accepted with probability $\exp(-\frac{\Delta cost}{T})$. Hence, at high temperatures, the algorithm searches the solution space quite globally and at low temperatures, it searches around the local minimum.

3 Encoding Using Simulated Annealing

In this section we describe our implementation of simulated annealing to find an optimal solution to the BDD encoding problem.

3.1 Algorithm

We assume that logarithmic encoding is used for all states, which means that we use the smallest number of bits required to encode the states. A code where all the bits are either 0 or 1 is called a *code point*, e.g., if the number of bits used is 3, then 010 is a code point and 01- is not. The initial move randomly assigns a code point to each state. If the number of states is not a power of two, then some code points are not used. The starting temperature is 100. The temperature is reduced by a constant factor of 0.8, i.e., $T = 0.8T$. The stopping criterion for each temperature is when the number of consecutively rejected moves is 3. The next move function is a swap of code points between two randomly chosen states if the number of states is a power of two. If it is not, then the next move function is a swap of the code points between two randomly chosen states or a swap of the code point of a randomly chosen state and a randomly chosen unused code point.

We perform our experiment for both the functional and the relational representations of FSMs. For either of these representations, we build the BDDs for each encoding in each move. The number of BDD nodes is our cost function.

For incompletely specified FSMs, all unspecified transitions are treated as no change in state. In other words, for a present state and primary inputs combination, the next state is the same as the present state if the transition is not specified.

It is well known that variable ordering affects the BDD size. In this experiment, we consider several variable orderings for the relational representation. In our variable orderings, when we say that the present state and next state variables are interleaved, we mean that the i-th present state variable is immediately followed by the i-th next state variable in the ordering.

The orderings from the lowest level to the highest level follow (note that the lower is the level of a variable, the higher is its position in the BDD):

- Ordering **I**: inputs, present states, next states, outputs.
- Ordering **II**: inputs, present states, next states, outputs. The present state and next state variables are interleaved.
- Ordering **III**: inputs, outputs, present states, next states.
- Ordering **IV**: inputs, outputs, present states, next states. The present state and next state variables are interleaved.

To see the effect of outputs and next states in the monolithic relation representation, we also perform an experiment where we build two BDDs, one for the characteristic function of the primary inputs, present states, and next states combination, the other one for the characteristic function of the primary inputs, present states, and primary outputs combination. For ease of comparison, we also call these two variations as orderings:

- Ordering **V**: inputs, present states, next states.
- Ordering **VI**: inputs, present states, next states. The present state and next state variables are interleaved.
- Ordering **VII**: inputs, present states, outputs.

3.2 Experimental Results

Our implementation uses the Long's BDD package. The experiments were performed on a DEC AlphaServer 8400 5/300 with 2Gb of memory.

Our test cases include the MCNC benchmark set. The simulated annealing algorithm is run once for each circuit. The results of the simulated annealing runs for CSFSMs are tabulated in Table 1 through Table 7.

Table 1: Statistics for Simulated Annealing Runs for Completely Specified FSMs with Ordering I: Inputs, Present States, Next States, Outputs.

Name	Min	Max	Ave	Std Dev	Max-Min	CPU Time
dk15x	80	86	83	2	6	27.369
dk17x	75	89	84	3	14	59.758
ellen.min	93	93	93	0	0	135.905
ellen	113	125	124	2	12	66.277
ex6inp	167	190	180	5	23	40.990
fsm1	45	53	50	2	8	7.750
fstate	176	200	189	5	24	6.496
fsync	67	73	70	2	6	55.137
maincont	108	121	115	2	13	122.648
mc	53	56	55	1	3	12.593
ofsync	67	73	70	2	6	55.973
pkheader	20305	21585	20670	429	1280	196.632
scud	303	377	342	15	74	103.931
shiftreg	45	45	45	0	0	80.221
tav	116	116	116	0	0	293.309
tbk	493	552	530	13	59	3156.060
tbk_m	230	246	241	2	16	3092.892
virmach	462	473	468	3	11	65.961
vmecont	4238	4565	4392	66	327	6929.002

Table 2: Statistics for Simulated Annealing Runs for Completely Specified FSMs with Ordering II: Inputs, Present States and Next States Interleaved, Outputs.

Name	Min	Max	Ave	Std Dev	Max-Min	CPU Time
dk15x	76	88	83	4	12	22.174
dk17x	79	103	91	4	24	36.537
ellen.min	75	93	87	4	18	14.339
ellen	105	159	140	8	54	44.239
ex6inp	189	231	211	7	42	61.727
fsm1	44	56	49	2	12	15.886
fstate	177	214	188	9	37	2.937
fsync	66	75	71	3	9	56.903
maincont	104	132	117	5	28	91.178
mc	51	58	56	3	7	8.777
ofsync	66	75	71	3	9	61.492
pkheader	20051	22857	21218	954	2806	262.823
scud	407	505	455	18	98	125.601
shiftreg	23	45	39	3	22	9.484
tav	106	116	110	4	10	29.729
tbk	552	695	645	24	143	2709.038
tbk_m	244	267	259	4	23	3598.614
virmach	474	488	482	5	14	76.874
vmecont	3997	4197	4092	53	200	1172.417

Table 3: Statistics for Simulated Annealing Runs for Completely Specified FSMs with Ordering III: Inputs, Outputs, Present States, Next States.

Name	Min	Max	Ave	Std Dev	Max-Min	CPU Time
dk15x	93	99	96	2	6	15.758
dk17x	81	98	91	3	17	23.834
ellen.min	77	93	89	3	16	7.704
ellen	110	167	144	9	57	11.417
ex6inp	287	304	297	3	17	44.632
fsm1	42	50	46	1	8	6.246
fstate	220	228	225	2	8	20.866
fsync	92	98	96	2	6	25.700
maincont	116	126	123	2	10	48.793
mc	73	75	74	1	2	5.861
ofsync	92	98	96	2	6	25.672
pkheader	12475	12490	12484	3	15	8184.627
scud	582	609	598	5	27	175.837
shiftreg	27	45	40	3	18	3.509
tav	72	72	72	0	0	130.780
tbk	584	664	633	15	80	958.291
tbk_m	301	312	308	2	11	1177.322
virmach	654	660	657	2	6	40.692
vmecont	4999	5034	5016	6	35	6376.398

Table 4: Statistics for Simulated Annealing Runs for Completely Specified FSMs with Ordering IV: Inputs, Outputs, Present States and Next States Interleaved.

Name	Min	Max	Ave	Std Dev	Max-Min	CPU Time
dk15x	95	103	99	3	8	14.775
dk17x	87	106	96	3	19	27.489
ellen.min	73	93	88	3	20	8.116
ellen	90	156	138	11	66	1.575
ex6inp	297	321	309	4	24	37.290
fsm1	40	54	49	2	14	6.527
fstate	219	232	226	2	13	35.105
fsync	96	101	98	2	5	24.399
maincont	115	138	128	4	23	41.306
mc	73	75	74	1	2	5.802
ofsync	96	101	98	2	5	24.387
pkheader	12473	12497	12485	4	24	7125.212
scud	600	644	629	7	44	120.760
shiftreg	27	45	40	3	18	2.856
tav	70	72	71	1	2	27.405
tbk	668	774	717	18	106	662.522
tbk_m	298	328	320	4	30	851.905
virmach	656	663	659	2	7	41.140
vmecont	5016	5064	5039	10	48	4715.590

Table 5: Statistics for Simulated Annealing Runs for Completely Specified FSMs with Ordering V: Inputs, Present States, Next States.

Name	Min	Max	Ave	Std Dev	Max-Min	CPU Time
dk15x	19	23	21	1	4	11.542
dk17x	41	51	46	2	10	19.673
ellen.min	21	29	27	2	8	4.475
ellen	49	61	60	2	12	15.522
ex6inp	68	92	82	4	24	16.385
fsm1	32	39	36	1	7	4.428
fstate	81	101	89	5	20	8.791
fsync	24	28	26	2	4	13.122
mc	20	23	21	1	3	2.704
ofsync	24	28	26	2	4	13.119
pkheader	48	53	51	1	5	4508.126
scud	189	240	214	10	51	48.259
shiftreg	21	29	27	2	8	3.437
tav	9	9	9	0	0	76.352
tbk	381	435	416	9	54	805.450
tbk_m	167	178	174	2	11	723.678
virmach	97	103	100	2	6	13.344
vmecont	234	269	256	5	35	3126.848
maincont	76	84	81	2	8	32.197
shift4	47	61	59	2	14	12.574

Table 6: Statistics for Simulated Annealing Runs for Completely Specified FSMs with Ordering VI: Inputs, Present States and Next States Interleaved.

Name	Min	Max	Ave	Std Dev	Max-Min	CPU Time
shift4	50	72	64	3	22	10.475
dk15x	22	27	24	2	5	13.949
dk17x	47	62	54	2	15	19.784
ellen.min	16	34	29	3	18	5.272
ellen	31	71	64	3	40	11.498
ex6inp	85	115	103	5	30	14.310
fsm1	33	43	38	2	10	4.567
fstate	87	127	102	7	40	16.528
fsync	25	33	29	3	8	14.744
maincont	75	98	87	4	23	33.172
mc	19	25	23	3	6	2.980
ofsync	25	33	29	3	8	14.753
pkheader	47	63	55	2	16	2768.773
scud	217	287	258	14	70	55.968
shiftreg	16	34	29	3	18	4.721
tav	6	9	7	1	3	22.303
tbk	513	615	570	20	102	748.454
tbk_m	197	224	214	4	27	808.002
virmach	100	107	103	2	7	12.131
vmecont	289	343	316	9	54	2519.523

Table 7: Statistics for Simulated Annealing Runs for Completely Specified FSMs with Ordering VII: Inputs, Present States, Outputs.

Name	Min	Max	Ave	Std Dev	Max-Min	CPU Time
shift4	8	16	13	1	8	11.932
dk15x	47	53	50	3	6	10.446
dk17x	32	42	37	2	10	16.608
ellen.min	45	45	45	0	0	39.095
ellen	49	61	60	2	12	18.589
ex6inp	130	151	141	4	21	18.239
fsm1	16	21	19	1	5	4.491
fstate	55	55	55	0	0	122.558
fsync	40	43	41	1	3	22.281
maincont	31	35	34	1	4	29.012
mc	37	40	39	1	3	4.517
ofsync	40	43	41	1	3	22.267
pkheader	9778	10286	10032	192	508	54.751
scud	264	338	303	15	74	28.346
shiftreg	3	6	5	1	3	3.982
tav	57	57	57	0	0	104.691
tbk	99	120	110	4	21	1372.429
tbk_m	80	83	83	1	3	1239.739
virmach	439	452	446	4	13	20.705
vmecont	2260	2320	2302	10	60	4380.045

For comparing ordering I – IV, we summarize the data into Table 8. Columns 2 through 5 list the minimum numbers of BDD nodes in each ordering. Columns 6 through 9 show the average BDD sizes. The standard deviations are listed from column 10 through 13.

Table 8: Simulated annealing runs for relational representation of CSFSMs.

Name	Min BDD Size				Ave BDD Size			
	I	II	III	IV	I	II	III	IV
dk15x	80	76	93	95	83	83	96	99
dk17x	75	79	81	87	84	91	91	96
ellen.min	93	75	77	73	93	87	89	88
ellen	113	105	110	90	124	140	144	138
ex6inp	167	189	287	297	180	211	297	309
fstate	176	177	220	219	189	188	225	226
fsync	67	66	92	96	70	71	96	98
maincont	108	104	116	115	115	117	123	128
mc	53	51	73	73	55	56	74	74
ofsync	67	66	92	96	70	71	96	98
pkheader	20305	20051	12475	12473	20670	21218	12484	12485
scud	303	407	582	600	342	455	598	629
shiftreg	45	23	27	27	45	39	40	40
tav	116	106	72	70	116	110	72	71
tbk	493	552	584	668	530	645	633	717
tbk_m	230	244	301	298	241	259	308	320
virmach	462	474	654	656	468	482	657	659
vmecont	4238	3997	4999	5016	4392	4092	5016	5039

Our results show that for CSFSMs, interleaving present state and next state variables increases or decreases the BDD sizes by only a small amount. We see that Ordering I and II are generally better than Ordering III and IV. We also found that different encodings do not change dramatically the BDD size representing a CSFSM.

If we normalized the results with respect to Ordering I, we get Table 9.

Table 9: Comparison of Simulated Annealing Runs for Completely Specified FSMs with Different Ordering.

Name	in-ps-ns-out (I)	in-ps-ns-int-out (II)	in-out-ps-ns (III)	in-out-ps-ns-int (IV)
dk15x	1.00	0.95	1.16	1.19
dk17x	1.00	1.05	1.08	1.16
ellen.min	1.00	0.81	0.83	0.78
ellen	1.00	0.93	0.97	0.80
ex6inp	1.00	1.13	1.72	1.78
fsm1	1.00	0.98	0.93	0.89
fstate	1.00	1.01	1.25	1.24
fsync	1.00	0.99	1.37	1.43
maincont	1.00	0.96	1.07	1.06
mc	1.00	0.96	1.38	1.38
ofsync	1.00	0.99	1.37	1.43
pkheader	1.00	0.99	0.61	0.61
scud	1.00	1.34	1.92	1.98
shiftreg	1.00	0.51	0.60	0.60
tav	1.00	0.91	0.62	0.60
tbk	1.00	1.12	1.18	1.35
tbk_m	1.00	1.06	1.31	1.30
virmach	1.00	1.03	1.42	1.42
vmecont	1.00	0.94	1.18	1.18

The results for the ISFSMs are tabulated in Table 10 through Table 16.

Table 10: Statistics for Simulated Annealing Runs for Incompletely Specified FSMs with Ordering I: Inputs, Present States, Next States, Outputs.

Name	Min	Max	Ave	Std Dev	Max-Min	CPU Time
bbara	81	96	91	2	15	31.702
bbgun	6565	7444	6746	172	879	693.835
bbsse	275	328	299	9	53	33.760
bbtas	46	50	49	1	4	10.008
beecount	63	71	67	1	8	8.547
cf	783	930	853	45	147	2.738
chanstb	235	240	238	2	5	51.266
cpab	850	920	884	22	70	2.496
cse	299	335	319	6	36	58.540
dec	8213	8878	8550	228	665	62.087
dk14x	133	168	155	7	35	16.372
dk16x	316	343	332	5	27	68.159
dol2	33	40	36	2	7	4.623
donfile	156	176	166	4	20	47.585
es	33	36	35	1	3	2.542
ex1inp	1570	1690	1626	33	120	21.914
ex2inp	156	182	171	4	26	31.885
ex2out	99	115	108	2	16	18.354
ex3inp	85	100	94	3	15	7.359
ex3out	39	40	40	0	1	5.044
ex4inp	198	234	211	6	36	12.930
ex5inp	82	105	96	4	23	12.098
ex5out	32	33	33	0	1	4.251
ex7inp	89	107	99	3	18	15.528
ex7out	30	31	30	0	1	4.207
fs1	577	645	598	8	68	820.313
keyb	304	345	324	8	41	127.162
kirkman	1164	1239	1205	17	75	80.454
lion	22	24	23	1	2	2.774
lion9	67	87	77	3	20	6.113
mark1	190	211	197	6	21	11.304
master	77247	80495	78515	995	3248	43.800
modulo12	52	59	57	1	7	7.526
opus	128	149	141	4	21	17.458
p21stg	5268	6002	5585	253	734	34.096
planet	2098	2179	2131	32	81	5.914
pma	667	758	713	16	91	136.658
ricks	1344	1937	1453	131	593	10.329
rpss	3531	3801	3591	64	270	48.280
s1	829	963	885	36	134	10.476
s1a	721	858	780	36	137	9.756
s298_m	1	1	1	0	0	1.216
s8	40	47	43	2	7	6.662
sand	3473	3941	3598	195	468	16.751
saucier	938	1092	993	43	154	6.233
scf	165369	166737	165754	519	1368	1110.547
scf_m	166071	167329	166187	306	1258	867.329
slave	1232	1436	1334	41	204	94.487
str	1747	2084	1859	86	337	37.401
styr	805	929	869	23	124	193.404
tlc34stg	559	634	601	12	75	2605.982
tma	329	376	357	9	47	26.088
tr4	647	692	676	10	45	76.659
train11	74	89	83	2	15	11.410
viterbi	10602	11150	10722	195	548	76.348

Table 11: Statistics for Simulated Annealing Runs for Incompletely Specified FSMs with Ordering II: Inputs, Present States and Next States Interleaved, Outputs.

Name	Min	Max	Ave	Std Dev	Max-Min	CPU Time
bbara	85	116	103	5	31	21.956
bbgun	7040	7979	7355	387	939	160.726
bbsse	304	385	342	16	81	10.505
bbtas	43	53	49	2	10	8.248
beecount	62	77	71	2	15	12.359
cf	803	1033	882	51	230	8.615
chanstb	240	247	244	2	7	45.449
cpab	866	1076	927	37	210	3.269
cse	355	416	389	11	61	33.374
dec	9636	9834	9716	81	198	16.340
dk14x	128	195	162	12	67	21.917
dk16x	294	324	311	8	30	2.111
dol2	35	49	42	3	14	2.999
donfile	177	220	198	7	43	29.595
es	33	36	35	1	3	3.875
ex1inp	1652	1813	1697	50	161	9.973
ex2inp	177	216	198	6	39	24.569
ex2out	102	128	116	4	26	13.129
ex3inp	88	114	102	4	26	16.092
ex3out	38	40	39	1	2	3.662
ex4inp	197	267	221	12	70	14.495
ex5inp	85	116	103	5	31	10.536
ex5out	30	32	31	1	2	3.013
ex7inp	88	118	105	5	30	9.760
ex7out	30	31	31	0	1	3.300
fs1	526	588	559	18	62	5.736
keyb	374	492	428	24	118	67.678
kirkman	1082	1198	1155	40	116	4.933
lion	23	29	26	2	6	1.858
lion9	66	101	82	6	35	5.736
mark1	177	217	196	12	40	0.690
master	78600	83176	80589	1719	4576	42.074
modulo12	42	61	54	3	19	9.034
opus	136	190	165	8	54	12.104
p21stg	7193	8260	7754	339	1067	41.413
planet	2138	2389	2258	70	251	30.336
pma	710	757	733	15	47	4.018
ricks	1352	1805	1680	179	453	4.410
rpss	3626	4245	3715	139	619	27.667
s1	1157	1349	1242	65	192	1.504
s1a	966	1160	1047	71	194	1.301
s298_m	1	1	1	0	0	1.219
s8	42	56	49	3	14	4.191
sand	3806	4647	4057	290	841	15.897
saucier	1027	1258	1090	58	231	2.473
scf	165825	167183	166314	482	1358	1230.330
scf_m	166339	166815	166491	98	476	1627.056
slave	1316	1632	1453	69	316	27.604
str	2311	2447	2371	59	136	3.137
styr	932	1022	956	27	90	2.962
tlc34stg	622	732	672	21	110	1189.675
tma	308	379	344	11	71	39.150
tr4	579	674	634	24	95	107.805
train11	70	94	81	5	24	5.701
viterbi	10757	10875	10807	37	118	59.098

Table 12: Statistics for Simulated Annealing Runs for Incompletely Specified FSMs with Ordering III: Inputs, Outputs, Present States, Next States.

Name	Min	Max	Ave	Std Dev	Max-Min	CPU Time
bbara	84	97	92	2	13	32.291
bbgun	10250	10333	10290	16	83	> 3600
bbsse	428	463	446	6	35	34.467
bbtas	37	41	39	1	4	10.629
beecount	66	76	71	2	10	17.652
cf	1164	1194	1178	5	30	152.494
chanstb	290	292	291	1	2	59.682
cpab	1220	1291	1247	18	71	4.246
cse	497	552	529	10	55	90.142
dec	32499	32561	32528	12	62	> 3600
dk14x	155	182	171	5	27	24.772
dk16x	295	339	310	9	44	6.304
dol2	35	42	38	2	7	5.720
donfile	158	178	168	4	20	56.214
es	32	34	33	1	2	4.373
ex1inp	1470	1515	1491	8	45	519.462
ex2inp	172	198	187	4	26	33.198
ex2out	112	130	122	3	18	11.932
ex3inp	91	103	97	2	12	8.194
ex3out	36	40	38	1	4	4.802
ex4inp	242	262	254	3	20	31.985
ex5inp	93	117	104	4	24	14.449
ex5out	36	39	38	1	3	3.678
ex7inp	98	113	106	3	15	7.609
ex7out	32	33	32	0	1	5.023
fs1	561	633	588	8	72	994.974
keyb	554	671	610	21	117	97.369
kirkman	668	686	678	4	18	529.587
lion	22	28	25	2	6	2.641
lion9	67	90	80	4	23	9.217
mark1	424	472	436	9	48	31.556
master	229775	229903	229815	38	128	388.482
modulo12	52	59	57	1	7	9.238
opus	182	203	195	3	21	22.869
p21stg	5385	5856	5607	169	471	60.905
planet	3626	3737	3675	16	111	674.982
pma	831	858	846	4	27	178.521
ricks	1443	1452	1449	1	9	452.581
rpss	9261	9323	9281	11	62	505.975
s1	994	1062	1028	13	68	104.700
s1a	1362	1499	1421	36	137	15.392
s298_m	1	1	1	0	0	1.219
s8	42	49	45	2	7	7.056
sand	5554	5634	5590	14	80	937.464
saucier	2051	2108	2073	10	57	82.516
scf	spaceout	spaceout	spaceout	spaceout	spaceout	82.516
scf_m	spaceout	spaceout	spaceout	spaceout	spaceout	82.516
slave	5467	5540	5497	14	73	279.029
str	2223	2255	2241	5	32	304.214
styr	1342	1501	1417	33	159	105.743
tlc34stg	643	713	678	13	70	1552.804
tma	382	411	402	5	29	34.343
tr4	1010	1057	1044	10	47	151.329
train11	72	92	83	3	20	11.344
viterbi	122902	123085	122982	43	183	> 3600

Table 13: Statistics for Simulated Annealing Runs for Incompletely Specified FSMs with Ordering IV: Inputs, Outputs, Present States and Next States Interleaved.

Name	Min	Max	Ave	Std Dev	Max-Min	CPU Time
bbara	87	114	102	4	27	31.887
bbgun	10136	10234	10198	13	98	> 3600
bbsse	438	499	465	9	61	42.206
bbtas	35	45	41	2	10	7.099
beecount	69	84	78	2	15	19.353
cf	1173	1230	1201	9	57	173.077
chanstb	290	292	291	1	2	60.329
cpab	1234	1294	1266	20	60	2.739
cse	533	604	575	13	71	79.043
dec	32482	32563	32521	16	81	> 3600
dk14x	166	197	183	6	31	25.082
dk16x	307	361	335	9	54	34.719
dol2	37	50	44	3	13	6.179
donfile	182	219	202	6	37	39.491
es	31	34	33	1	3	3.921
ex1inp	1478	1553	1522	13	75	407.393
ex2inp	198	241	220	8	43	19.174
ex2out	118	147	133	5	29	13.447
ex3inp	97	121	110	4	24	16.421
ex3out	38	44	41	2	6	3.274
ex4inp	241	265	253	5	24	30.567
ex5inp	98	137	118	6	39	10.804
ex5out	37	40	39	1	3	3.677
ex7inp	106	138	120	5	32	10.794
ex7out	32	35	34	1	3	4.038
fs1	523	584	555	18	61	6.538
keyb	595	775	685	31	180	93.184
kirkman	661	691	675	5	30	746.609
lion	22	30	27	3	8	1.137
lion9	57	100	85	6	43	10.515
mark1	420	491	445	15	71	27.253
master	229829	230013	229894	44	184	429.335
modulo12	45	60	54	2	15	9.154
opus	184	212	199	5	28	12.922
p21stg	6984	7751	7407	236	767	46.584
planet	3640	3738	3700	19	98	93.083
pma	821	875	852	8	54	136.640
ricks	1438	1454	1447	3	16	228.394
rpss	9279	9370	9311	15	91	627.444
s1	1071	1143	1101	19	72	8.235
s1a	1607	1800	1688	71	193	2.019
s298_m	1	1	1	0	0	1.217
s8	44	57	51	3	13	7.519
sand	5561	5680	5620	21	119	840.860
saucier	2048	2130	2090	16	82	44.791
scf	spaceout	spaceout	spaceout	spaceout	spaceout	44.791
scf_m	spaceout	spaceout	spaceout	spaceout	spaceout	44.791
slave	5499	5627	5542	24	128	188.113
str	2204	2252	2233	8	48	293.552
styr	1419	1656	1499	55	237	15.369
tlc34stg	729	802	769	14	73	1122.563
tma	377	428	407	8	51	27.037
tr4	984	1069	1040	15	85	158.463
train11	70	98	86	4	28	11.073
viterbi	122906	123068	122995	37	162	923.282

Table 14: Statistics for Simulated Annealing Runs for Incompletely Specified FSMs with Ordering V: Inputs, Present States, Next States.

Name	Min	Max	Ave	Std Dev	Max-Min	CPU Time
bbara	66	82	77	3	16	23.678
bbgun	3138	3860	3427	216	722	37.029
bbsse	110	144	129	6	34	23.544
bbtas	23	28	27	1	5	5.850
beecount	38	48	43	2	10	13.799
cf	426	521	463	30	95	2.761
chanstb	85	91	88	2	6	31.421
cpab	346	371	359	6	25	4.791
cse	151	183	169	5	32	59.206
dec	7441	8119	7787	228	678	42.354
dk14x	47	64	56	3	17	18.621
dk16x	161	182	174	4	21	39.616
dol2	31	38	34	2	7	4.389
donfile	154	174	164	4	20	43.985
es	18	20	19	1	2	2.619
ex1inp	333	411	374	15	78	85.703
ex2inp	125	146	136	4	21	28.554
ex2out	74	86	81	2	12	16.830
ex3inp	60	74	69	2	14	10.485
ex3out	16	18	17	1	2	4.256
ex4inp	67	80	73	3	13	11.119
ex5inp	55	71	65	3	16	6.056
ex5out	17	20	19	1	3	2.602
ex7inp	59	73	67	2	14	9.537
ex7out	15	18	17	1	3	2.527
fs1	536	612	565	9	76	537.412
keyb	260	307	286	9	47	107.335
kirkman	56	56	56	0	0	1780.716
lion	15	19	17	2	4	1.968
lion9	54	72	64	3	18	9.917
mark1	90	112	97	5	22	10.838
master	15182	16049	15616	219	867	15.910
modulo12	50	57	55	1	7	7.398
opus	84	105	95	4	21	13.515
p21stg	5268	5848	5599	171	580	29.811
planet	475	518	495	12	43	20.908
pma	418	497	458	15	79	62.287
ricks	110	134	120	4	24	49.944
rpss	1676	1789	1710	34	113	20.064
s1	714	851	773	36	137	7.340
s1a	714	851	773	36	137	6.534
s298_m	1	1	1	0	0	1.207
s8	38	45	41	2	7	6.510
sand	2753	3275	2895	186	522	17.076
saucier	835	1061	916	49	226	7.336
scf	164011	165079	164311	384	1068	1950.843
scf_m	163971	165641	164379	378	1670	914.728
slave	662	779	723	29	117	5.931
str	1206	1519	1321	102	313	8.963
styr	446	558	508	19	112	20.277
tlc34stg	348	401	380	10	53	1457.302
tma	148	195	175	7	47	31.640
tr4	200	252	237	12	52	59.405
train11	58	76	68	2	18	10.635
viterbi	8849	9281	8928	151	432	40.705

Table 15: Statistics for Simulated Annealing Runs for Incompletely Specified FSMs with Ordering VI: Inputs, Present States and Next States Interleaved.

Name	Min	Max	Ave	Std Dev	Max-Min	CPU Time
bbara	72	101	88	4	29	14.815
bbgun	3221	4575	3463	318	1354	80.585
bbsse	131	198	162	11	67	13.318
bbtas	22	33	28	2	11	7.889
beecount	38	54	48	2	16	12.391
cf	451	639	505	41	188	4.853
chanstb	88	94	91	2	6	30.662
cpab	360	445	380	17	85	6.948
cse	194	245	221	9	51	35.684
dec	8917	9387	9033	161	470	12.229
dk14x	53	78	69	4	25	16.536
dk16x	190	230	210	6	40	43.976
dol2	33	46	40	3	13	4.748
donfile	178	215	198	6	37	31.345
es	19	21	20	1	2	3.140
ex1inp	365	507	437	24	142	92.557
ex2inp	143	181	163	6	38	26.144
ex2out	79	100	90	3	21	14.717
ex3inp	68	90	80	4	22	11.778
ex3out	19	22	21	1	3	4.184
ex4inp	66	99	81	6	33	6.384
ex5inp	60	85	75	4	25	7.471
ex5out	19	21	20	1	2	2.707
ex7inp	66	88	78	4	22	14.230
ex7out	17	19	18	1	2	3.070
fs1	504	565	536	18	61	5.308
keyb	332	437	377	19	105	72.528
kirkman	31	58	52	2	27	386.493
lion	15	22	19	3	7	2.198
lion9	55	81	68	4	26	5.320
mark1	88	133	106	9	45	10.903
master	15688	16682	16106	336	994	9.874
modulo12	43	58	52	2	15	7.218
opus	84	131	109	7	47	11.920
p21stg	7021	8050	7636	302	1029	34.351
planet	563	673	608	22	110	40.495
pma	427	529	488	21	102	26.007
ricks	112	150	127	6	38	49.761
rpss	1754	2034	1839	93	280	9.788
s1	959	1152	1040	71	193	0.984
s1a	959	1152	1040	71	193	0.869
s298_m	1	1	1	0	0	1.195
s8	40	53	47	3	13	6.968
sand	3088	3837	3273	260	749	13.656
saucier	881	1183	993	65	302	4.571
scf	164205	166234	164664	617	2029	2238.268
scf_m	164831	166908	165229	596	2077	998.619
slave	711	866	801	39	155	6.234
str	1796	1932	1856	60	136	1.669
styr	557	766	670	42	209	40.870
tlc34stg	420	534	490	19	114	1465.110
tma	167	220	193	10	53	21.687
tr4	222	293	266	16	71	25.158
train11	55	83	69	4	28	8.925
viterbi	8964	9780	9137	261	816	44.569

Table 16: Statistics for Simulated Annealing Runs for Incompletely Specified FSMs with Ordering VII: Inputs, Present States, Outputs.

Name	Min	Max	Ave	Std Dev	Max-Min	CPU Time
bbara	26	35	31	2	9	22.288
bbgun	2938	2951	2945	2	13	> 3600
bbsse	167	211	183	7	44	24.014
bbtas	20	23	22	1	3	7.572
beecount	31	36	34	1	5	17.204
cf	322	441	346	27	119	9.417
chanstb	47	48	47	0	1	51.949
cpab	40	46	43	1	6	67.931
cse	153	187	173	5	34	64.824
dec	432	441	437	2	9	522.087
dk14x	65	81	74	3	16	25.017
dk16x	61	76	69	3	15	31.844
dol2	5	7	6	1	2	7.223
donfile	9	17	14	1	8	35.077
es	17	19	18	1	2	3.274
ex1inp	1122	1325	1223	48	203	41.631
ex2inp	44	61	54	3	17	27.693
ex2out	32	44	39	2	12	16.530
ex3inp	28	37	34	2	9	14.782
ex3out	15	21	17	2	6	3.134
ex4inp	114	141	123	4	27	11.776
ex5inp	29	41	36	2	12	12.291
ex5out	16	17	17	0	1	4.172
ex7inp	30	43	39	2	13	12.063
ex7out	15	17	16	1	2	2.843
fs1	36	49	46	2	13	598.106
keyb	186	233	212	7	47	86.063
kirkman	711	811	772	23	100	64.560
lion	11	12	12	0	1	2.973
lion9	24	34	30	2	10	10.553
mark1	80	85	84	1	5	22.214
master	50020	51074	50512	297	1054	29.664
modulo12	4	11	9	1	7	6.265
opus	85	107	97	4	22	9.651
p21stg	46	61	55	3	15	1383.091
planet	1459	1646	1525	52	187	54.155
pma	342	401	372	10	59	115.791
ricks	1166	1533	1338	129	367	4.108
rpss	1195	1254	1210	12	59	24.627
s1	729	863	785	36	134	9.011
s1a	15	23	20	1	8	53.244
s298_m	1	1	1	0	0	1.205
s8	12	14	13	1	2	8.937
sand	1748	1999	1806	79	251	6.783
saucier	425	530	462	22	105	3.016
scf	spaceout	spaceout	spaceout	spaceout	spaceout	3.016
scf_m	spaceout	spaceout	spaceout	spaceout	spaceout	3.016
slave	1070	1279	1161	42	209	32.244
str	395	397	397	0	2	210.790
styr	438	528	490	17	90	151.965
tlc34stg	61	71	67	2	10	2220.255
tma	128	159	146	5	31	25.615
tr4	318	322	321	1	4	136.262
train11	26	36	32	2	10	9.699
viterbi	10037	10392	10144	114	355	69.893

For comparing ordering I – IV, we summmarize the data into Table 17. The entry "spaceout" means out of memory, and "> 3600" means stopped after the cpu time exceeds 3600 seconds.

Our results show that Ordering I and II are better in most cases. In some cases like *bbgun*, *dec*, and *viterbi*, they are substantially better. The large discrepancy in BDD sizes for these cases is due to the large BDD needed to represent the primary outputs. Interestingly, BDD sizes are smaller when state variables are not interleaved. Different encodings do affect the BDD sizes of ISFSMs more than those of CSFSMs; however, the differences are not substantial.

If we normalized the results with respect to Ordering I, we get Table 18.

For functional representations, the results are shown in Table 19. Our results show that even for CSFSMs, different encodings affect the BDD size considerably for some circuits. For example, the average size for *maincont* is 90 with a standard deviation of 12, while the minimum BDD size that the simulated annealing algorithm found is 43. This also means that there are not many encodings which would produce small BDDs for this circuit.

The results for ISFSMs are shown in Table 20. We see from this table that, similarly to CSFSMs, encoding plays an important role in determining the BDD size. For example, the minimum BDD size for *dec* is 9212, while the average size is 13090 with a standard deviation of 2017 nodes.

4 Exact Algorithm

To evaluate the effectiveness of our simulated annealing algorithm, we need to model the BDD encoding problem and provide an exact algorithm that solves it. Since this is a complex problem, we model a restricted version of the problem, namely the *BDD input encoding problem*. The reason for which we are looking into this problem is that it can be shown that the optimum solution of the BDD input encoding problem yields the optimum relational representation of an FSM as long as the state variables are not interleaved in the ordering. In the section, we provide a formal definition of this problem followed by an exact algorithm.

4.1 BDD Input Encoding Problem

We define the BDD input encoding problem as follows:

<u>**Input**</u>:

1. A set of symbolic values, $D = \{0, 1, 2, \ldots, |D| - 1\}$, where $|D| = 2^s$, for some $s \in \mathcal{N}$. A symbolic variable, v, taking values in D.
2. A set of symbolic values, $R = \{0, 1, 2, \ldots, |R| - 1\}$.
3. A set of functions, $F = \{f_0, f_1, f_2, \ldots, f_{|F|-1}\}$, where $f_i : D \mapsto R$.
4. A set of s binary variables, $B = \{b_{s-1}, b_{s-2}, \ldots, b_0\}$.

<u>**Output**</u>:

Bijection $e : D \mapsto \mathbf{B}^s$ such that the size of the BDD representing $e(F)$ is minimum, where $e(F) = \{e(f_0), e(f_1), \ldots, e(f_{|F|-1})\}$, and $e(f_i) : \mathbf{B}^s \mapsto R$. We call e an encoding of v and of F interchangeably. We call e_{opt} an encoding e that minimizes the size of the BDDs of $e(F)$, i.e., $e_{opt} = \min_e\{|e(F)|\}$, where $|e(F)|$ is the number of nodes of the multi-rooted BDD representing $e(F)$.

In other words, the problem is about finding an encoding of a multi-valued variable v such that the multi-rooted multi-terminal BDD representing a set of multi-valued functions of v has minimum number of nodes. Diagramatically, a multi-valued function f of v is represented as a single level multi-way tree. The root is labeled with v. A mapping $f(d) = r$ is represented by an edge labeled with d going from the root to a leaf node labeled with r. We call this diagram a *single level multi valued tree (SLMVT)*. For clarity purposes, the leaf nodes are replaced by their labels in all figures. An example of an SLMVT is the functions f and g shown in Figure 1.

Table 17: Simulated annealing runs for relational representation of ISFSMs.

Name	Min BDD Size				Ave BDD Size			
	I	II	III	IV	I	II	III	IV
bbara	81	85	84	87	91	103	92	102
bbgun	6565	7040	10250	10136	6746	7355	10290	10198
bbsse	275	304	428	438	299	342	446	465
bbtas	46	43	37	35	49	49	39	41
beecount	63	62	66	69	67	71	71	78
cf	783	803	1164	1173	853	882	1178	1201
chanstb	235	240	290	290	238	244	291	291
cpab	850	866	1220	1234	884	927	1247	1266
cse	299	355	497	533	319	389	529	575
dec	8213	9636	32499	32482	8550	9716	32528	32521
dk14x	133	128	155	166	155	162	171	183
dk16x	316	294	295	307	332	311	310	335
dol2	33	35	35	37	36	42	38	44
donfile	156	177	158	182	166	198	168	202
es	33	33	32	31	35	35	33	33
ex1inp	1570	1652	1470	1478	1626	1697	1491	1522
ex2inp	156	177	172	198	171	198	187	220
ex2out	99	102	112	118	108	116	122	133
ex3inp	85	88	91	97	94	102	97	110
ex3out	39	38	36	38	40	39	38	41
ex4inp	198	197	242	241	211	221	254	253
ex5inp	82	85	93	98	96	103	104	118
ex5out	32	30	36	37	33	31	38	39
ex7inp	89	88	98	106	99	105	106	120
ex7out	30	30	32	32	30	31	32	34
fs1	577	526	561	523	598	559	588	555
keyb	304	374	554	595	324	428	610	685
kirkman	1164	1082	668	661	1205	1155	678	675
lion	22	23	22	22	23	26	25	27
lion9	67	66	67	57	77	82	80	85
mark1	190	177	424	420	197	196	436	445
master	77247	78600	229775	229829	78515	80589	229815	229894
modulo12	52	42	52	45	57	54	57	54
opus	128	136	182	184	141	165	195	199
p21stg	5268	7193	5385	6984	5585	7754	5607	7407
planet	2098	2138	3626	3640	2131	2258	3675	3700
pma	667	710	831	821	713	733	846	852
ricks	1344	1352	1443	1438	1453	1680	1449	1447
rpss	3531	3626	9261	9279	3591	3715	9281	9311
s1	829	1157	994	1071	885	1242	1028	1101
s1a	721	966	1362	1607	780	1047	1421	1688
s298_m	1	1	1	1	1	1	1	1
s8	40	42	42	44	43	49	45	51
sand	3473	3806	5554	5561	3598	4057	5590	5620
saucier	938	1027	2051	2048	993	1090	2073	2090
scf	165369	165825	spaceout	spaceout	165754	166314	spaceout	spaceout
scf_m	166071	166339	spaceout	spaceout	166187	166491	spaceout	spaceout
slave	1232	1316	5467	5499	1334	1453	5497	5542
str	1747	2311	2223	2204	1859	2371	2241	2233
styr	805	932	1342	1419	869	956	1417	1499
tlc34stg	559	622	643	729	601	672	678	769
tma	329	308	382	377	357	344	402	407
tr4	647	579	1010	984	676	634	1044	1040
train11	74	70	72	70	83	81	83	86
viterbi	10602	10757	122902	122906	10722	10807	122982	122995

Table 18: Comparison of Simulated Annealing Runs for Incompletely Specified FSMs with Different Ordering.

Name	in-ps-ns-out	in-ps-ns-int-out	in-out-ps-ns	in-out-ps-ns-int
bbara	1.00	1.05	1.04	1.07
bbgun	1.00	1.07	1.56	1.54
bbsse	1.00	1.11	1.56	1.59
bbtas	1.00	0.93	0.80	0.76
beecount	1.00	0.98	1.05	1.10
cf	1.00	1.03	1.49	1.50
chanstb	1.00	1.02	1.23	1.23
cpab	1.00	1.02	1.44	1.45
cse	1.00	1.19	1.66	1.78
dec	1.00	1.17	3.96	3.95
dk14x	1.00	0.96	1.17	1.25
dk16x	1.00	0.93	0.93	0.97
dol2	1.00	1.06	1.06	1.12
donfile	1.00	1.13	1.01	1.17
es	1.00	1.00	0.97	0.94
ex1inp	1.00	1.05	0.94	0.94
ex2inp	1.00	1.13	1.10	1.27
ex2out	1.00	1.03	1.13	1.19
ex3inp	1.00	1.04	1.07	1.14
ex3out	1.00	0.97	0.92	0.97
ex4inp	1.00	0.99	1.22	1.22
ex5inp	1.00	1.04	1.13	1.20
ex5out	1.00	0.94	1.12	1.16
ex7inp	1.00	0.99	1.10	1.19
ex7out	1.00	1.00	1.07	1.07
fs1	1.00	0.91	0.97	0.91
keyb	1.00	1.23	1.82	1.96
kirkman	1.00	0.93	0.57	0.57
lion	1.00	1.05	1.00	1.00
lion9	1.00	0.99	1.00	0.85
mark1	1.00	0.93	2.23	2.21
master	1.00	1.02	2.97	2.98
modulo12	1.00	0.81	1.00	0.87
opus	1.00	1.06	1.42	1.44
p21stg	1.00	1.37	1.02	1.33
planet	1.00	1.02	1.73	1.73
pma	1.00	1.06	1.25	1.23
ricks	1.00	1.01	1.07	1.07
rpss	1.00	1.03	2.62	2.63
s1	1.00	1.40	1.20	1.29
s1a	1.00	1.34	1.89	2.23
s298_m	1.00	1.00	1.00	1.00
s8	1.00	1.05	1.05	1.10
sand	1.00	1.10	1.60	1.60
saucier	1.00	1.09	2.19	2.18
scf	1.00	1.00	0.00	0.00
scf_m	1.00	1.00	0.00	0.00
slave	1.00	1.07	4.44	4.46
str	1.00	1.32	1.27	1.26
styr	1.00	1.16	1.67	1.76
tlc34stg	1.00	1.11	1.15	1.30
tma	1.00	0.94	1.16	1.15
tr4	1.00	0.89	1.56	1.52
train11	1.00	0.95	0.97	0.95
viterbi	1.00	1.01	11.59	11.59

Table 19: Simulated annealing runs for functional representation of CSFSMs.

Name	Min BDD Size	Ave BDD Size	Standard Deviation
dk15x	38	39	1
dk17x	72	79	2
ellen.min	7	15	2
ellen	21	41	2
ex6inp	86	108	8
fsm1	25	30	1
fstate	44	77	12
fsync	61	62	0
maincont	43	90	19
mc	16	17	1
ofsync	61	62	0
scud	200	254	19
shiftreg	5	14	2
tav	23	23	0
tbk	391	438	15
tbk_m	199	211	4
virmach	71	80	6
vmecont	365	398	13
pkheader	64	76	4

In this way we model the process of encoding the present state variables of a completely specified finite state machine (CSFSM) when the characteristic function of the CSFSM is represented by a BDD. We assume that the state variables are not interleaved in the variable ordering. In this respect, $f_i(d_i) = r_i$ represents the state transition from the present state d_i to the next state r_i under the proper input combination that causes this transition. Essentially, we cut across the BDDs representing the characteristic functions of CSFSMs and only look at the present state variables. Therefore, although the encoded BDDs are actually multi-terminal BDDs (MTBDDs), we still refer to them as BDDs. It is worth mentioning here that our formulation can also be applied to other BDD encoding problems, like MDD encoding and BDD re-encoding.

We assume that the BDDs are represented by their *true* edges. We do not model yet the complemented edges.

4.2 Characterization of BDD Node Reductions

Here we outline our strategy to find an optimum encoding. Assume that we have binary decision trees representing an instance of the BDD input encoding problem. The BDD representing this instance of the problem is obtained by applying the two BDD reduction rules, i.e.,

Rule 1: eliminating nodes with the same *then* and *else* children, and

Rule 2: eliminating duplicate isomorphic subgraphs.

We would like to characterize the conditions in the original problem where these rules can be applied. To apply these BDD reduction rules there must exist some isomorphic subgraphs. This means that there is a set of values in R where each value is incident to more than one edge in the SLMVT representing F. So from F, D, and R, we can group those edges into sets. Each group represents a possible reduction. However, there are many isomorphic subgraphs, therefore, applying a reduction to one may interfere with applying a reduction to the other. We therefore find the sets whose reductions do not interfere with each other and yield the largest possible total reduction. Once these are found, we encode each set in such a way that it occupies a subtree in the BDD representing $e(F)$.

With this characterization, we explain why encoding e_2 is better than encoding e_1 for the example in Figure 1.

1. $f(2) = f(4) = 0$. One node is reduced when 2 is encoded as 000 and 4 as 001.

Table 20: Simulated annealing runs for functional representation of ISFSMs.

Name	Min BDD Size	Ave BDD Size	Standard Deviation
bbara	73	84	3
bbgun	3706	4069	388
bbsse	135	173	14
bbtas	17	21	1
beecount	47	54	3
cf	324	446	50
chanstb	76	83	6
cpab	169	278	58
cse	190	221	14
dec	9212	13090	2017
dk14x	62	71	2
dk16x	151	167	5
dol2	18	25	2
donfile	111	127	6
es	16	18	1
ex1inp	491	569	56
ex2inp	103	122	5
ex2out	61	69	3
ex3inp	52	61	3
ex3out	17	18	1
ex4inp	69	101	15
ex5inp	41	55	4
ex5out	14	16	1
ex7inp	45	59	3
ex7out	12	15	1
fs1	699	741	32
keyb	394	483	48
kirkman	403	415	3
lion	10	12	1
lion9	40	49	3
mark1	73	94	7
master	3545	4244	387
modulo12	22	32	2
opus	106	137	11
p21stg	8631	9031	317
planet	1374	1452	65
pma	1182	1273	40
ricks	186	246	24
rpss	879	990	159
s1	1155	1297	125
s1a	940	1075	93
s298_m	1	1	0
s8	44	52	2
sand	2314	2884	328
saucier	412	485	48
scf	68798	84517	22819
scf_m	24748	43165	14430
slave	823	971	90
str	1042	1333	169
styr	710	751	59
tlc34stg	427	479	21
tma	207	251	15
tr4	200	226	9
train11	42	53	3
viterbi	2431	3006	309

2. $f(0) = g(0) = 1$ and $f(3) = g(3) = 2$. Encoding 0 as 010 and 3 as 011 allows sharing of one node between f and g, i.e., the subtree identified by the cube 01-.

3. $f(1) = g(7) = 3$ and $f(6) = g(5) = 4$. Encoding 1 as 100, 6 as 101, 7 as 110, and 5 as 111 allows sharing of one node between f and g, i.e., the subtree of encoded f identified by 10- or the subtree of encoded g identified by 11-.

4. $f(7) = g(1) = 5$ and $f(5) = g(6) = 6$. Using the same encoding as in 3 allows sharing of one node between f and g, i.e., the subtree of encoded f identified by 11- or the subtree of encoded g identified by 10-.

Equivalently, encoding e_2 allows us to apply the two BDD reduction rules, namely, eliminating a node with same children and eliminating isomorphic subgraphs; while encoding e_1 does not.

4.2.1 Sibling and Isomorphic Sets

The objective of this section is to identify all cases where Rule 1 and Rule 2 can be applied. For that, we define two sets, the *sibling set* and the *isomorphic set*. Intuitively, we are trying to capture in the sibling sets the conditions where Rule 1 can be applied, and in the isomorphic sets the conditions where Rule 2 can be applied. Informally, each element of a sibling set S is a 2-tuple (l^0, l^1) where l^0 and l^1 are ordered sets of symbolic values that can be encoded so that they share an isomorphic subgraph and the isomorphic subgraph is both the *then* child and the *else* child of a node (i.e., the only child). An isomorphic set I is a collection of ordered sets l of symbolic values that can be encoded so that all ordered sets share an isomorphic subgraph.

As examples, consider the four SLMVTs shown in Figure 5. There are 8 edges in each of the four cases shown. All edges not shown are assumed to point to values other than 0 and 1. The variables needed to encode these cases are b_2, b_1, and b_0 in that order. The binary decision trees representing an optimum solution for each case are shown in Figure 6. The corresponding BDDs are shown in Figure 7. We show for these examples and for these optimum encodings the relation of isomorphic subgraphs versus sibling and isomorphic sets.

1. Case (a): Since $f(0) = f(1)$, there is an encoding (e.g., $e(0) = 000$, $e(1) = 001$) such that in the encoded BDD there is a node (i.e., n_2) whose edges point to the same node (and so can be reduced). This fact is captured by $S_0 = \{(0_f), (1_f)\}$. Similarly for S_1, replacing "$f(0) = f(1)$" with "$f(2) = f(3)$".

   Since $f(0) = f(2)$ and $f(1) = f(3)$, there is an encoding (e.g., $e(0) = 000$, $e(1) = 001$, $e(2) = 010$, $e(3) = 011$) such that in the encoded BDD there are nodes (i.e., n_2 and n_3) with isomorphic subgraphs (and so can be reduced). This fact is captured by $I_0 = \{(0_f, 1_f), (2_f, 3_f)\}$.

   Since $f(0) = f(2)$ and $f(1) = f(3)$, there is an encoding (e.g., $e(0) = 000$, $e(1) = 001$, $e(2) = 010$, $e(3) = 011$) such that in the encoded BDD there are nodes (i.e., n_2 and n_3) with isomorphic subgraphs (and so can be reduced) and a node (i.e., n_1) whose edges point to the same node. This fact is captured by $S_2 = \{(0_f, 1_f), (2_f, 3_f)\}$. We would like to point out that although this constraint also captures the previous constraint, it is used differently. I_0 is to capture Rule 2 and S_2 is to capture Rule 1. Both are needed to calculate correctly the number of nodes that can be reduced by an encoding later.

2. Case (b): Since $f(0) = f(1)$, there is an encoding (e.g., $e(0) = 000$, $e(1) = 001$) such that in the encoded BDD there is a node (i.e., n_2) whose edges point to the same node (and so can be reduced). This fact is captured by $S_0 = \{(0_f), (1_f)\}$. Similarly for S_1, replacing "$f(0) = f(1)$" with "$f(2) = f(3)$".

   Since $f(0) = f(1)$ and $f(2) = f(3)$, there is an encoding (e.g., $e(0) = 000$, $e(1) = 010$, $e(2) = 001$, $e(3) = 011$) such that in the encoded BDD there are nodes (i.e., n_2 and n_3 of Figure 8) with isomorphic subgraphs (and so can be reduced). This fact is captured by $I_0 = \{(0_f, 2_f), (1_f, 3_f)\}$.

Since $f(0) = f(1)$ and $f(2) = f(3)$, there is an encoding (e.g., $e(0) =$ 000, $e(1) =$ 010, $e(2) =$ 001, $e(3) =$ 011) such that in the encoded BDD there are nodes (i.e., n_2 and n_3 of Figure 8) with isomorphic subgraphs (and so can be reduced) and a node (i.e., n_1 of Figure 8) whose edges point to the same node. This fact is captured by $S_2 = \{(0_f, 2_f), (1_f, 3_f)\}$.

For this case, the encoding induced by S_0 and S_1 can not satisfy the encoding induced by I_0 and S_2, and vice versa. This leads to our notion of compatibility below. An encoding that satisfies S_0 and S_1 is $e(0) =$ 000, $e(1) =$ 001, $e(2) =$ 010, $e(3) =$ 011 and the BDD is shown in Figure 7. An encoding that satisfies I_0 and S_2 is $e(0) =$ 000, $e(1) =$ 010, $e(2) =$ 001, $e(3) =$ 011 and the BDD is shown in Figure 8.

3. Case (c): Since $f(0) = g(0)$ and $f(1) = g(1)$, there is an encoding (e.g., $e(0) =$ 000, $e(1) =$ 001) such that in the encoded BDD there are nodes (i.e., m_2 and n_2) with isomorphic subgraphs (and so can be reduced). This fact is captured by $I_0 = \{(0_f, 1_f), (0_g, 1_g)\}$.
4. Case (d): Since $f(0) = g(2)$ and $f(1) = g(3)$, there is an encoding (e.g., $e(0) =$ 000, $e(1) =$ 001, $e(2) =$ 010, $e(3) =$ 011) such that in the encoded BDD there are nodes (i.e., m_2 and n_2) with isomorphic subgraphs (and so can be reduced). This fact is captured by $I_0 = \{(0_f, 1_f), (2_g, 3_g)\}$.

For each case above, there are other sibling and isomorphic sets. We will show how to construct the complete collection of all sibling and isomorphic sets later.

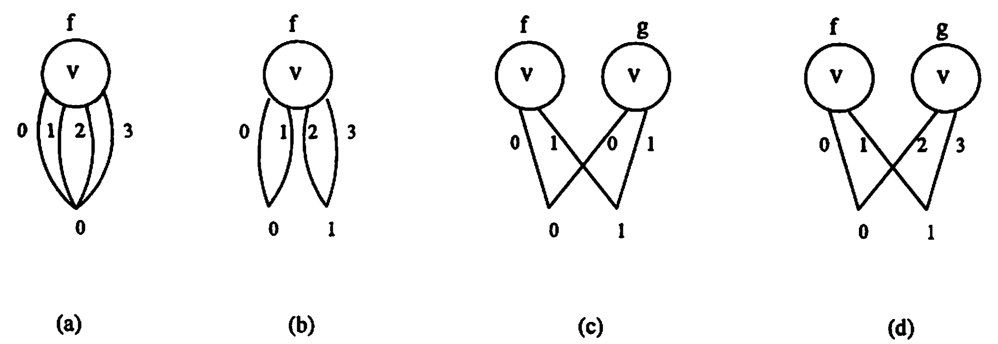


Figure 5: Examples of sibling and isomorphic sets.

Formally, sibling and isomorphic sets are defined as follows.

Definition 2 *A labeled symbol d_f has a symbol $d \in D$ and a label $f \in F$. It is the d-edge of the SLMVT representing f. The following notations are defined for d_f: $sym(d_f) = d$, $fn(d_f) = f$, and $val(d_f) = f(d)$.*

Definition 3 *A symbolic list l is an ordered set (or list) of labeled symbols with no duplicate and all labeled symbols have the same function. The k-th element of l is denoted as l_k. The set of all symbols of l is $Sym(l) = \{sym(l_k) \mid 0 \leq k \leq |l| - 1\}$. The function of l is $Fn(l) = fn(l_0)$.*

Definition 4 *An isomorphic set I is a set of at least two symbolic lists. The j-th element of I is denoted as l^j. I satisfies the following three conditions:*

1. *The sizes of all symbolic lists of I are the same and they are a power of two, i.e., $\exists a \in \mathcal{N}\ \forall l \in I\ (|l| = 2^a)$.*
2. *The k-th elements of all symbolic lists of I have the same value, i.e., $\exists r_k \in R\ \forall l \in I\ (val(l_k) = r_k)$, $0 \leq k \leq |l| - 1$.*

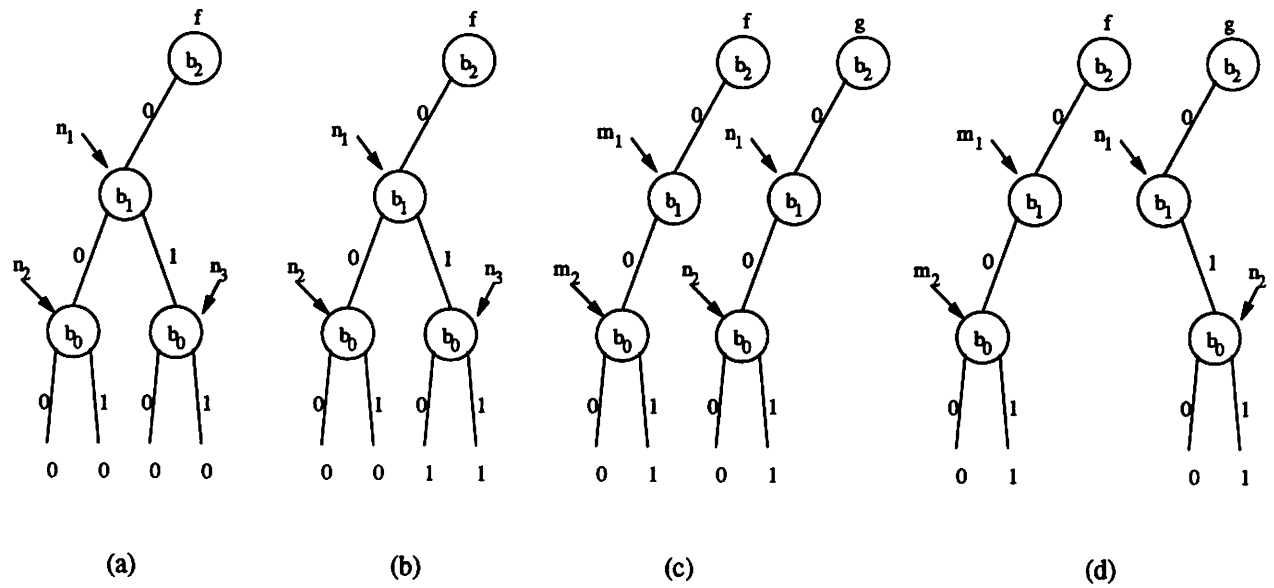


Figure 6: Binary Decision Trees for an Optimum Encoding.

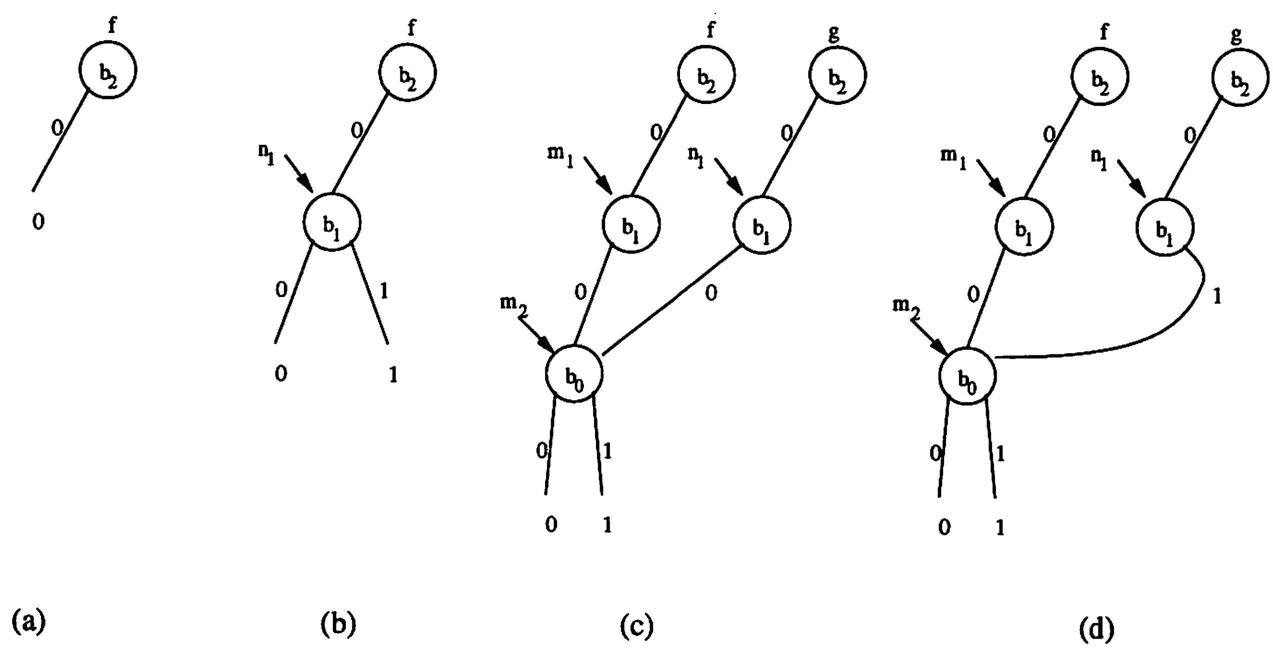


Figure 7: BDDs for an Optimum Encoding.

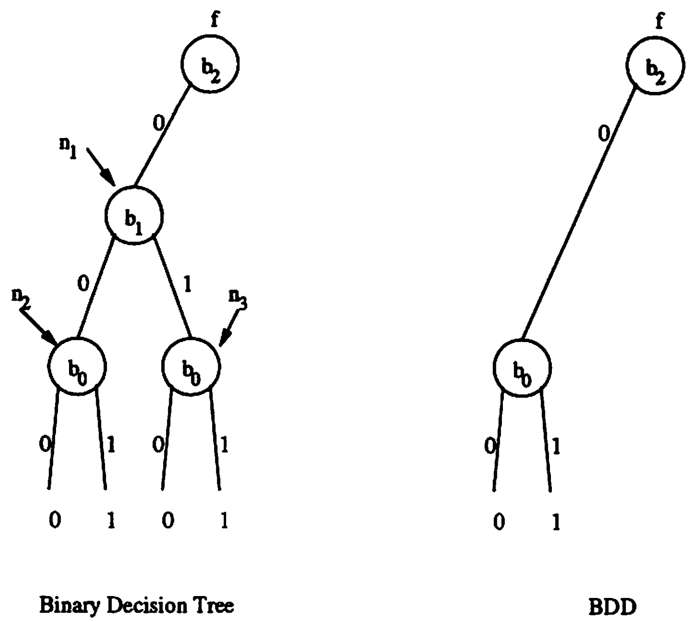


Figure 8: Alternative Optimum Encoding for Case (b).

3. *For any two lists $l', l'' \in I$, either for every index k the symbols of the k-th elements of l' and l'' are the same or the symbol of no element of l' is the same as the symbol of an element of l'', i.e., $\forall l' \in I\ \forall l'' \in I\ ((\forall k\ sym(l'_k) = sym(l''_k)) \vee (\forall i\ \forall j\ sym(l'_i) \neq sym(l''_j)))$, $0 \leq i,j,k \leq |l'| - 1$.*

Definition 5 *A sibling set S is an isomorphic set with 2 symbolic lists, l^0 and l^1, and satisfies the following conditions:*

1. *The symbol of no element of l^0 is the same as the symbol of an element of l^1, i.e., $\forall i\ \forall j\ (sym(l^0_i) \neq sym(l^1_j)), 0 \leq i,j \leq |l^0| - 1$.*
2. *The functions of l^0 and l^1 are the same, i.e., $Fn(l^0) = Fn(l^1)$.*

We will see that for every sibling set there is an equivalent isomorphic set. For an instance of the BDD input encoding problem, the set of all sibling sets is denoted as $\mathcal{S}$, and the set of all isomorphic sets is denoted as $\mathcal{I}$.

In the following discussion, the term *tree* is used to mean the encoded binary tree representing a function.

Definition 6 *Given an encoding e and a set of symbols $D' \subseteq D$, the tree* spanned *by the codes of the symbols in D' is the tree T whose root is the least common ancestor of the terminal nodes of the codes of the symbols in D'. Furthermore, every leaf of T is the code of a symbol in D'. We say also that D' spans T (denoted by $T_{D'}$).*

For example, given the problem in Figure 1 and the encoding e_2 as in page 2, the codes for the symbols 0 and 3 span the tree rooted at T in Figure 4.

Proposition 1 *Given a sibling set $S = \{l^0, l^1\}$, there is an encoding e such that the codes of the symbols in $l^0 \cup l^1$ span exactly a tree whose root has a left subtree spanned exactly by the symbols in l^0 and a right subtree spanned exactly by the symbols in l^1.*

Proof: Let the size of both l^0 and l^1 be 2^a. Let $b(i)$ denote the a-bit binary number representing integer i. Then symbols in l^0 and l^1 are encoded as $e(sym(l_i^0)) = y0b(i)$, and $e(sym(l_i^1)) = y1b(i)$, $0 \leq i \leq |l^0| - 1$, where y is an arbitrary 0-1 string of $s-(a+1)$ bits. This encoding exists always because the symbols of l^0 and l^1 are by definition disjoint. With this encoding, the proposition follows. ■

Proposition 2 *Given an isomorphic set $I = \{l^i\}, 0 \leq i \leq |I| - 1$, there is an encoding e such that $\forall l^i \in I$ the symbols in l^i span exactly a subtree T_{l^i} and all T_{l^i}s are isomorphic.*

Proof: Let the size of all l^is be 2^a. Let $b(i)$ denote the a-bit binary number representing integer i. Then the symbols in all l^is are encoded as $e(sym(l_j^i)) = y_i b(j)$, $0 \leq i \leq |I| - 1$, $0 \leq j \leq |l^i| - 1$, where y_i is a string of $s - a$ bits and $y_i = y_{i'}$ if and only if $l^i = l^{i'}$. With this encoding, the proposition follows. ■

To illustrate these propositions, we look back to the example in Figure 1. The $\mathcal{S}$ and $\mathcal{I}$ of this example are:

1. $S_0 = \{(2_f), (4_f)\}$.
2. $I_0 = \{(0_f, 3_f), (0_g, 3_g)\}$, $I_1 = \{(3_f, 0_f), (3_g, 0_g)\}$.
3. $I_2 = \{(1_f, 6_f), (7_g, 5_g)\}$, $I_3 = \{(6_f, 1_f), (5_g, 7_g)\}$.
4. $I_4 = \{(7_f, 5_f), (1_g, 6_g)\}$, $I_5 = \{(5_f, 7_f), (6_g, 1_g)\}$.

For now, we focus only on S_0, I_0, I_2, and I_4. Each S_i or I_i justifies why encoding e_2 is better than encoding e_1 in this example. In other words, S_0, I_0, I_2, and I_4 contain requirements to find an optimum encoding. Following the proof of the above propositions, S_0 states that 2 and 4 should be encoded such that they differ only in b_0 to span a subtree and save a node. I_0 states that 0 and 3 should be encoded such that they differ only in b_0 for symbols in I_0 to span a subtree and share a node. I_2 states not only that 1 and 6 should be encoded such that they differ only in b_0, and similarly for 7 and 5, but also that the value of b_0 of 1 should be the same as the value of b_0 of 7 and the value of b_0 of 6 should be the same as the value of b_0 of 5 for symbols in I_2 to span isomorphic subtrees and share a node. I_4 essentially states the same requirements as I_2. All of these requirements are satisfied by encoding e_2, but not by e_1.

4.2.2 Finding $\mathcal{S}$ and $\mathcal{I}$

Given an instance of the BDD input encoding problem, we show an algorithm that finds the sets $\mathcal{S}$ and $\mathcal{I}$.

Algorithm 1 finds for each subset R' of R, the set of all possible symbolic lists whose values are exactly R'. For example, for the SLMVT of Figure 1, the algorithm finds the following:

Values	Symbolic Lists
0	$\{2_f\}, \{4_f\}, \{2_f, 4_f\}$
1	$\{0_f\}, \{0_g\}$
2	$\{3_f\}, \{3_g\}$
⋮	⋮

Algorithm 2 uses shorter symbolic lists to construct larger lists. On the same example, the algorithm generates:

0, 1	$\{0_f, 2_f\}, \{0_f, 4_f\}, \{0_f, 2_f, 4_f\}$
0, 2	$\{2_f, 3_f\}, \{3_f, 4_f\}, \{2_f, 3_f, 4_f\}$
⋮	⋮
0, 1, 2	$\{0_f, 2_f, 3_f\}, \{0_f, 3_f, 4_f\}, \{0_f, 2_f, 3_f, 4_f\}$
⋮	⋮

Algorithm 3 then reads all the symbolic lists and generate all possible combinations of them to compute $\mathcal{S}$ and $\mathcal{I}$.

5 Algorithm to Generate $\mathcal{S}$ and $\mathcal{I}$

Algorithm 1 (Generating Symbolic Lists)

```
generateLists(F, D, R)
Input: An instance of the BDD input encoding problem, F, D, R.
Output: The set of all symbolic lists ℒ of F, D, R.

  ℒ = ∅
  for r = 1 to |R| do
    foreach f ∈ F do
      L = {d | f(d) = r}
      ℒ'[r] = {all subsets of L}
      ℒ ← ℒ ∪ ℒ'[r]
    end for
  end for

  k = 0
  while (ℒ' ≠ ∅) do
    for i = 1 to |ℒ'| − 1 do
      for j = i + 1 to |ℒ'| do
        ℒ''[k] = generateListsFromPair(ℒ'[i], ℒ'[j])
        ℒ ← ℒ ∪ ℒ''[k]
      end for
    end for
    ℒ' = ℒ''
  end while
```

Algorithm 2 (Generating Symbolic Lists from Shorter Symbolic Lists)

```
generateListsFromPair(ℒ1, ℒ2)
Input: Two sets of symbolic lists: ℒ1, ℒ2.
Output: The set ℒn of symbolic lists such that each symbolic list is the concatenation
        of a symbolic list in ℒ1 and another symbolic list in ℒ2.
Comment: append(l', l'') takes two symbolic lists l' and l'' as arguments and returns a
        list lexicographically ordered which is a concatenation of l' and l'' with duplicates
        removed if Fn(l') = Fn(l''). Otherwise, it returns an empty set.

  ℒn = ∅
  for i = 1 to |ℒ1| do
    for j = 1 to |ℒ2| do
      l = append(ℒ1[i], ℒ2[j])
      if l is a symbolic list then
        ℒ ← ℒ ∪ l
      end if
    end for
  end for
```

Algorithm 3 (Generating $\mathcal{S}$ and $\mathcal{I}$)

```
generateSets(F, D, R)
```

Input: An instance of the BDD input encoding problem, F, D, R.
Output: The sets $\mathcal{S}$ and $\mathcal{I}$.
Comment: $permute(\mathcal{S})$ takes a set of sibling sets or a set of isomorphic sets and appends to it all permutations of each sibling or isomorphic set.

```
𝓛 = generateLists(F, D, R)
for i = 1 to |𝓛| do
  L = 𝓛[i]
  for j = 1 to |L| do
    foreach L' of (|L| choose j) combinations of L do
      if all lists in L' form a sibling set S, then
        𝓢 ← 𝓢 ∪ S
      else if there is a permutation of lists in L' such
          that they form a sibling set S, then
        𝓢 ← 𝓢 ∪ S
      end if
      if all lists in L' form an isomorphic set I, then
        𝓘 ← 𝓘 ∪ I
      else if there is a permutation of lists in L' such
          that they form an isomorphic set I, then
        𝓘 ← 𝓘 ∪ I
      end if
    end for
  end for
end for
permute(𝓢)
permute(𝓘)
```

Having computed $\mathcal{S}$ and $\mathcal{I}$, we can state the following theorem.

Theorem 5.1 *Using only $\mathcal{S}$ and $\mathcal{I}$, an optimum encoding e_{opt} can be obtained.*

Proof:
Let T be the forest of binary decision trees representing $e_{opt}(F)$. To get from T the BDD representing $e_{opt}(F)$, the BDD reduction rules, Rule 1 and Rule 2, are applied. It suffices to prove that any reduction can be found by using only $\mathcal{S}$ and $\mathcal{I}$. We divide the proof into two parts, according to whether Rule 1 or Rule 2 are applied:

1. Applying Rule 1: Consider part of T in Figure 9. Let x_i be a node with label b_i. Assume that we can apply Rule 1 at x_i, then $then(x_i)$ is isomorphic with $else(x_i)$. Let an arbitrary path from a function f in $e_{opt}(F)$ to x_i be p_i. Also let t_k be the path from $then(x_i)/else(x_i)$ to leaf v_k, $0 \leq k \leq m-1$, where m is the number of leaves in the subtree rooted at $then(x_i)/else(x_i)$. Define symbolic lists
$$M_i = (d_0, d_1, \ldots, d_{m-1}),$$
where $sym(d_k) = e_{opt}^{-1}(p_i\overline{b_i}t_k)$, $fn(d_k) = f$, $val(d_k) = v_k$, and
$$M_i' = (d_0', d_1', \ldots, d_{m-1}'),$$
where $sym(d_k') = e_{opt}^{-1}(p_i b_i t_k)$, $fn(d_k') = f$, $val(d_k') = v_k$.
Then, we have:

(a) $|M_i|$ and $|M_i'|$ are equal and are powers of two,

(b) For any t_k, $f(p_i\overline{b_i}t_k) = f(p_ib_it_k) = v_k$.

$S = \{M_i, M_i'\}$ is a sibling set because:

- Property (a) is exactly condition 1 of Definition 4.
- Property (b) satisfies condition 2 of Definition 4 because all k-th elements of $|M_i|$ and $|M_i'|$ have the same value.
- Property (b) satisfies condition 3 of Definition 4 and condition 1 of Definition 5 because all elements of $|M_i|$ and $|M_i'|$ are different.
- By definition, both $|M_i|$ and $|M_i'|$ contain symbols from the same function; therefore condition 2 of Definition 5 is satisfied.

Moreover, *generateLists*(F, D, R) generates both $|M_i|$ and $|M_i'|$, and since S is a sibling set, *generateSets*(F, D, R) generates S. We will show later that an encoding that takes advantage of the reduction implied by S can be found.

2. Applying Rule 2: Consider the part of T in Figure 10. Let x_i and x_j be two nodes in T with labels b_i. Without loss of generality, assume that we can apply Rule 2 at $then(x_i)$ and $then(x_j)$, then $then(x_i)$ is isomorphic with $then(x_j)$ in the BDD representing $e_{opt}(F)$. Let p_i and p_j be two arbitrary paths in T from function f_i to x_i and function f_j to x_j, respectively. Also let t_k be the path from $then(x_i)/then(x_j)$ to leaf v_k, $0 \leq k \leq m-1$, where m is the number of leaves in the subtree rooted at $then(x_i)/then(x_j)$. Define symbolic lists

$$M_i = (d_0, d_1, \ldots, d_{m-1}),$$

where $sym(d_k) = e_{opt}^{-1}(p_ib_it_k)$, $fn(d_k) = f_i$, $val(d_k) = v_k$, and

$$M_j = (d_0', d_1', \ldots, d_{m-1}'),$$

where $sym(d_k') = e_{opt}^{-1}(p_jb_it_k)$, $fn(d_k') = f_j$, $val(d_k') = v_k$.

Then, we have:

(a) $|M_i|$ and $|M_j|$ are equal and are powers of two,

(b) For any t_k, $f_i(p_ib_it_k) = f_j(p_jb_it_k) = v_k$.

$I = \{M_i, M_j\}$ is an isomorphic set because:

- Property (a) is exactly condition 1 of Definition 4.
- Property (b) satisfies condition 2 of Definition 4 because all k-th elements of $|M_i|$ and $|M_i'|$ have the same value.
- If $p_i = p_j$, then all k-th symbols of M_i and $Sym(M_j)$ are the same, and if $p_i \neq p_j$, then no element of $Sym(M_i)$ is the same as any element of $Sym(M_j)$, and this satisfies condition 3 of Definition 4.

Moreover, *generateLists*(F, D, R) generates both $|M_i|$ and $|M_i'|$, and since I is an isomorphic set, *generateSets*(F, D, R) generates I. We will show later that an encoding that takes advantage of the reduction implied by I can be found. If there are more than two nodes where we can apply Rule 2, the set I would simply contain more elements.

Note that cases 1 and 2 are sufficient for this proof. All other reductions are just a combination of cases 1 and 2. For example, consider Figure 11, by case 2, there is an $I = \{l^0, l^1, \ldots, l^{|Q_i|-1}, l^{|Q_i|}\}$, where each list l^r, $r = 0, 1, \ldots, |Q_i| - 1, |Q_i|$ contains the symbols encoded by the minterms of paths passing through $then(x_r)$ and ending respectively, in the leaves of subtrees $T_0, T_1, \ldots, T_{Q_i-1}, T_{|Q_i|} = T_j$. By case 1, there exists an S for each node in the subtree Q_i, where Q_i is the subtree rooted at $then(x_i)$ and all leaves have labels b_j.

■

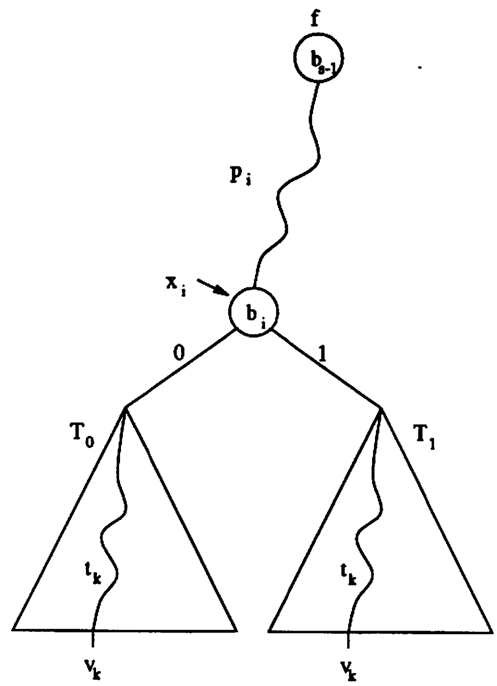


Figure 9: Binary Tree for Case 1 of the Proof of Theorem 5.1.

Theorem 5.1 says that $\mathcal{S}$ and $\mathcal{I}$ contain all the information that is needed to find an optimum encoding. The question now is to find a subset of $\mathcal{S}$ and $\mathcal{I}$ that corresponds to an optimum encoding. This is the topic of the next section.

5.1 Finding an Optimal Encoding

From here on, the number of nodes that can be reduced is with respect to the complete binary trees that represent the encoded F. When not specified, a set means either a sibling set or an isomorphic set.

5.1.1 Compatibility of Sibling and Isomorphic Sets

Sibling sets and isomorphic sets specify that if their symbols are encoded to satisfy the reductions implied, then Rule 1 and Rule 2 can be applied to merge isomorphic subgraphs and reduce nodes. Hence, they implicitly specify the number of nodes that can be reduced, which we refer to as *gains*.

Definition 7 *The gain of a sibling set S, denoted as $gain(S)$, is equal to 1. The gain of an isomorphic set I, denoted as $gain(I)$ is equal to $(|I| - 1) \times (|l^0| - 1)$, where $l^0 \in I$.*

$\mathcal{S}$ and $\mathcal{I}$ contain the information for all possible reductions. However, not all sets may be selected together. For example, the sibling set $S = \{(1_f), (2_f)\}$ and isomorphic set $I = \{(2_f, 3_f), (2_g, 3_g)\}$ of

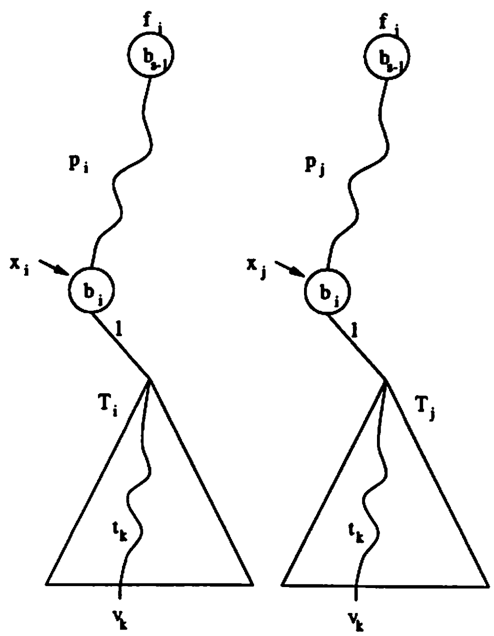


Figure 10: Binary Tree for Case 2 of the Proof of Theorem 5.1.

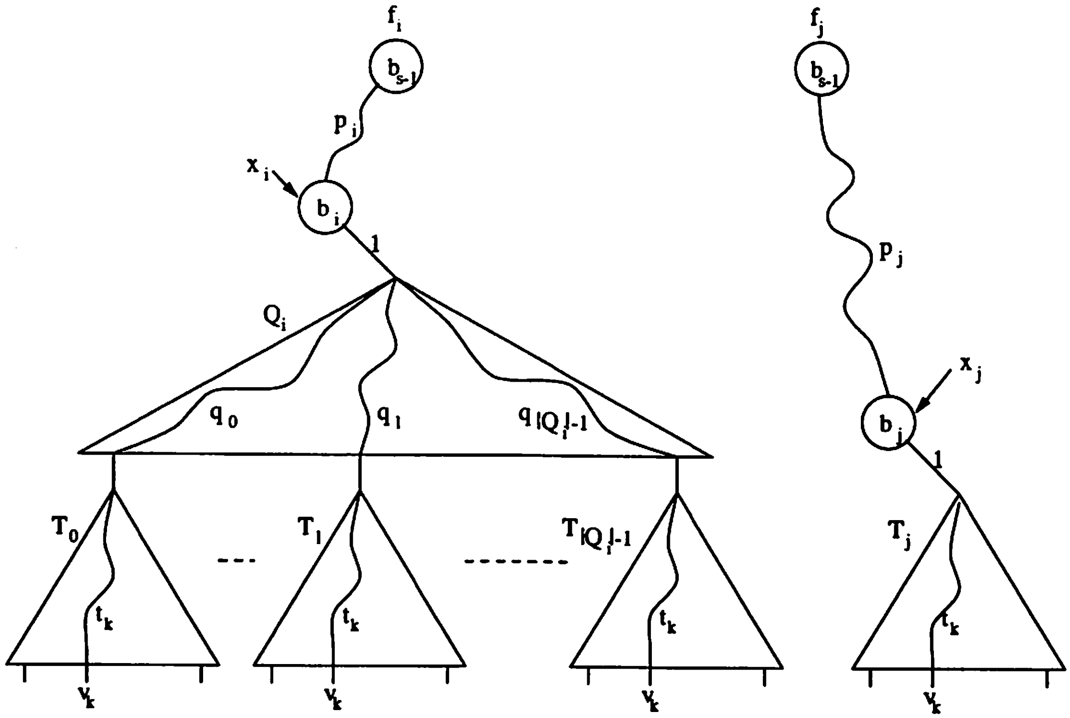


Figure 11: Binary Tree for Combination of Cases 1 and 2 of the Proof of Theorem 5.1.

Figure 12 can not be selected together because S says that symbols 1 and 2 should span exactly a subtree while I says that symbols 2 and 3 should span exactly a subtree. Hence, an encoding can only benefit from either S or I. We therefore need to identify which sets can be selected together and which can not. For that we define the notion of *compatibility*.

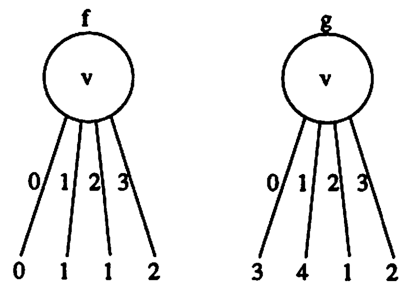


Figure 12: Example of Incompatible Sets

Definition 8 *A collection of sets $\mathcal{S}$ and $\mathcal{I}$ are compatible if there is an encoding e such that all reductions implied by the sets $S \in \mathcal{S}$ and $I \in \mathcal{I}$ can be applied to the complete binary decision tree yielded by e.*

Definition 9 *Symbolic lists l' and l'' are compatible, denoted as $l' \sim l''$, if one or more of the following conditions are true:*

1. $Sym(l') \cap Sym(l'') = \emptyset$, *i.e., the set of symbols of l' does not intersect the set of symbols of l''.*
2. $\exists a \in \mathcal{N}\ \forall k\ sym(l'_k) = sym(l''_{a|l'|+k})$, $0 \leq k \leq |l'| - 1$, $|l''| \geq (a+1) \times |l'|$, *i.e., the symbols of l' match exactly the symbols of l'' in the same order starting at position $a \times |l'|$.*
3. $\exists a \in \mathcal{N}\ \forall k\ sym(l''_k) = sym(l'_{a|l''|+k})$, $0 \leq k \leq |l''| - 1$, $|l'| \geq (a+1) \times |l''|$, *i.e., the symbols of l'' match exactly the symbols of l' in the same order starting at position $a \times |l''|$.*

Definition 9 says that two lists are compatible if their symbols do not intersect or the symbols of one list is a subset of the symbols of the other starting at a power of 2 position.

Definition 10 *Sibling list l^S of sibling set S is the symbolic list constructed by concatenating l^0 and l^1 of S.*

Theorem 5.2 *If l' and l'' are compatible, then there exists an encoding e such that the symbols of l' and l'' span exactly a subtree respectively.*

Proof: $l' \sim l''$ implies one of the following:

1. Every symbol of l' is different from any symbol of l''. In this case, there certainly exists an encoding such that the theorem is true.
2. The symbols of l' match exactly the symbols of l'' starting at position $a \times |l'|$ in the same order and $|l''| \geq (a+1) \times |l'|$. In this case we encode the symbols of l' and l'' such that the symbols of l'' span exactly a tree and the symbols of l' span exactly a subtree of the tree formed by the codes of the symbols of l''.
3. This case is the dual of case 2.

■

Theorem 5.3 *If a set L of symbolic lists are pair-wise compatible, then there exists an encoding e such that the symbols of every symbolic list in L span exactly a subtree.*

Proof: Since two symbolic lists are compatible if and only if their symbols do not overlap or the symbols of one are a sublist of those of the other starting at a power of 2 position, there is a notion of maximality in L. Sorting L in non-increasing order and applying the encoding procedure in the Proof of Theorem 5.2 produces the results satisfying the claim of this theorem. ∎

Theorem 5.4 *Sibling sets S' and S'' are compatible if $l^{S'}$ is compatible with $l^{S''}$.*

Proof: By Theorem 5.2, there exists an encoding e such that the symbols of $l^{S'}$ (consequently the symbols of S') and the symbols of $l^{S''}$ (consequently the symbols of S'') span exactly a subtree respectively. It follows that encoding e allows the reductions implied by S' and S'' to be applied. ∎

Theorem 5.5 *Sibling set S and isomorphic set I are compatible if l^S is compatible with every list of I.*

Proof: For any $l', l'' \in I$, either the k-th symbols of l' and l'' are the same or all symbols of l' and l'' are different. In the former case, we encode the symbols of either l', or l'' to span exactly a tree. In the latter case, we encode the symbols of l' to span a tree, and the symbols of l'' to span another tree. Then by Theorem 5.2, there exists an encoding e such that symbols of l^S and $l^i, 0 \leq i \leq |I| - 1$, span exactly a subtree respectively. It follows that encoding e allows the reductions implied by both S and I to be applied. ∎

Theorem 5.6 *Isomorphic sets I' and I'' are compatible if every list $l' \in I'$ is compatible with every list $l'' \in I''$.*

Proof: This proof is the same as the proof of Theorem 5.5 except that one argues on both I' and I''. ∎

A set of compatible sets is called a *compatible*. The compatibility of two sets simply means that the reduction induced by one set does not prevent that of the other. For example, it can be seen that S_0, I_0, I_2, and I_4 of Figure 1 are mutually compatible, and therefore form a compatible.

5.1.2 Encoding Sibling and Isomorphic Sets

We begin this section by stating the following corollary which follows immediately from the theorems in the previous section.

Corollary 5.1 *Given a compatible C, there exists an encoding e such that the reductions implied by all its elements can be applied.*

Definition 11 *Let X' be either a sibling or an isomorphic set and X'' another sibling or isomorphic set. Then X' is contained in X'' if $\forall l' \in X' \; \exists l'' \in X'' \; (l' \subset l'')$ and X' is completely contained in X'' if $\exists l'' \in X'' \; \forall l' \in X' \; (l' \subset l'')$.*

For example, the set $I_0 = \{(0_f, 1_f), (0_g, 1_g)\}$ is contained, but not completely contained in $I_1 = \{(0_f, 1_f, 2_f, 3_f), (0_g, 1_g, 2_g, 3_g)\}$; while $S_0 = \{(0_f), (1_f)\}$ is completely contained in I_1. This definition is used for gain calculation and encoding of a compatible. The motivation of this definition is that the reduction implied by I_0 is covered by I_1, but the reduction implied by S_0 is not. The gain of a compatible that contains only S_0, I_0, and I_1 is equal to the sum of the gains of S_0 and I_1 only.

Algorithm 4 computes the codes of a compatible. The idea is that starting with a binary tree, we assign codes to the symbols of symbolic lists by non-increasing length of the symbolic lists. The symbols of a symbolic list are assigned to occupy the largest subtree of codes still available.

Algorithm 4 (Encoding a Compatible)

encode(C, D)
Input: A compatible C, a set of symbols D.
Output: Codes for D stored in 2-dimensional array code.
Comment: *reverseBit*() takes an integer argument and reverses all its bits.
$c_{i,j}$ denotes the j-th element of the i-th list of c.
order is the array of ordered codes, e.g. 0000, 1000, 0100, 1100, 0010, 1010, ...
This array recursively partitions all the codes into two equal partitions and orders them in non-increasing size.

```
/* Initialize codes */
for d = 0 to |D| - 1 do
  for i = 0 to s - 1 do
    code[d][i] = '-'

/* Initialize orders */
for i = 0 to |D| - 1 do
  order[i] = reverseBit(i)

/* Get top level sets and sort them in non-increasing cube size */
Top = {c | c ∈ C and no d ∈ C contains c}
foreach c ∈ Top do
  if c is a sibling set
    cubeSize(c) = 2 × |l^0| of c
  else
    cubeSize(c) = |l^0| of c
T_sorted = sort T in non-increasing cubeSize

/* Encode sorted top level sets */
foreach c ∈ T_sorted do
  if (c is a sibling set and code[c_{0,0}][0] = '-') then
    while (code[order[j]][0] = '-') do
      j = j + 1
    for i = 0 to |c_0| - 1 do
      code[sym(c_0, i)] = order[j] + i
    for i = 0 to |c_1| - 1 do
      code[sym(c_1, i)] = order[j] + |c_0| + i
  else
    for i = 0 to |c| - 1 do
      if (code[c_{i,0}][0] = '-') then
        while (code[order[j]][0] = '-') do
          j = j + 1
        for k = 0 to |c_i| - 1 do
          code[sym(c_{i,k})] = order[j] + k

/* Encode remaining codes */
for d = 0 to |D| - 1 do
  if code[d] = '-' then
    while (code[order[j]][0] = '-') do
      j = j + 1
```

code[*d*] = *order*[*j*]
return code

5.1.3 Gain of a Compatible

Using Algorithm 4, an encoding that allows the reductions implied by all sibling and isomorphic sets of a compatible C can be found. We denote the encoding found by Algorithm 4 by $e_{alg}(C)$. Since there may exist many compatibles for an instance of the BDD input encoding problem, we would like to find a compatible implying the largest reduction. Hence, we need to calculate the number of nodes that are reduced by a compatible. We call this quantity the *gain* of a compatible.

Definition 12 *The gain of a compatible C is equal to the difference in the number of nodes of the binary decision trees representing F and the number of nodes of the BDDs representing F encoded by $e_{alg}(C)$.*

With this definition, the following theorem can be stated.

Theorem 5.7 *A compatible of maximum gain yields an optimal encoding.*

Proof: Suppose that the theorem is not true, then either one of the following must be true:

1. There exists a better encoding, but no compatible captures it. A better encoding in this case means that more reductions than those implied by any compatible can be applied. But by Theorem 5.1, we know that every reduction is modeled by either a sibling or an isomorphic set, and by definition of compatibility, reductions implied by two incompatible sets can not be applied together. Hence, all optimal encodings must be yielded by compatibles.

2. There exists another compatible with a lower gain that yields BDDs with fewer number of nodes. This is not possible, because by Definition 12, compatibles with larger gains yield smaller BDDs.

■

The task is then to find a compatible with the *largest* gain. Unlike the example in Figure 1, where the gain of the compatible formed by S_0, I_0, I_2, and I_4 is simply the sum of the individual gains of its elements, the gain of an arbitrary compatible is more complicated to calculate without actually building the BDDs. If we apply a reduction rule induced by a set, then this reduction causes a merging of two isomorphic subgraphs. For these two subgraphs, there may exist two identical reductions within them. The gain of these two reductions should only be counted once. An example of this kind is shown in Figure 13. The following sibling and isomorphic sets form a compatible:

$$\begin{aligned} S_0 &= \{(0_f),(1_f)\} \\ S_1 &= \{(0_g),(1_g)\} \\ I_0 &= \{(0_f,1_f),(0_g,1_g)\} \\ I_1 &= \{(2_f,3_f),(2_g,3_g)\} \\ I_2 &= \{(0_f,1_f,2_f,3_f),(0_g,1_g,2_g,3_g)\} \end{aligned}$$

The gain of this compatible is not the sum of the gains of its elements because the reductions implied by I_0 and I_1 and one of the reductions implied by S_0 and S_1 are subsumed by the reduction implied by I_2. Then the gain of this compatible is equal to $gain(I_2) + gain(S_0) = 3 + 1 = 4$.

The basic idea is to find the sets with largest lists, calculate their gains, remove all gains of lists that are counted more than once and remove all sets that are subsumed by other sets. The complete algorithm is listed in Algorithm 5.

Algorithm 5 (Gain Calculation)

Gain(C)

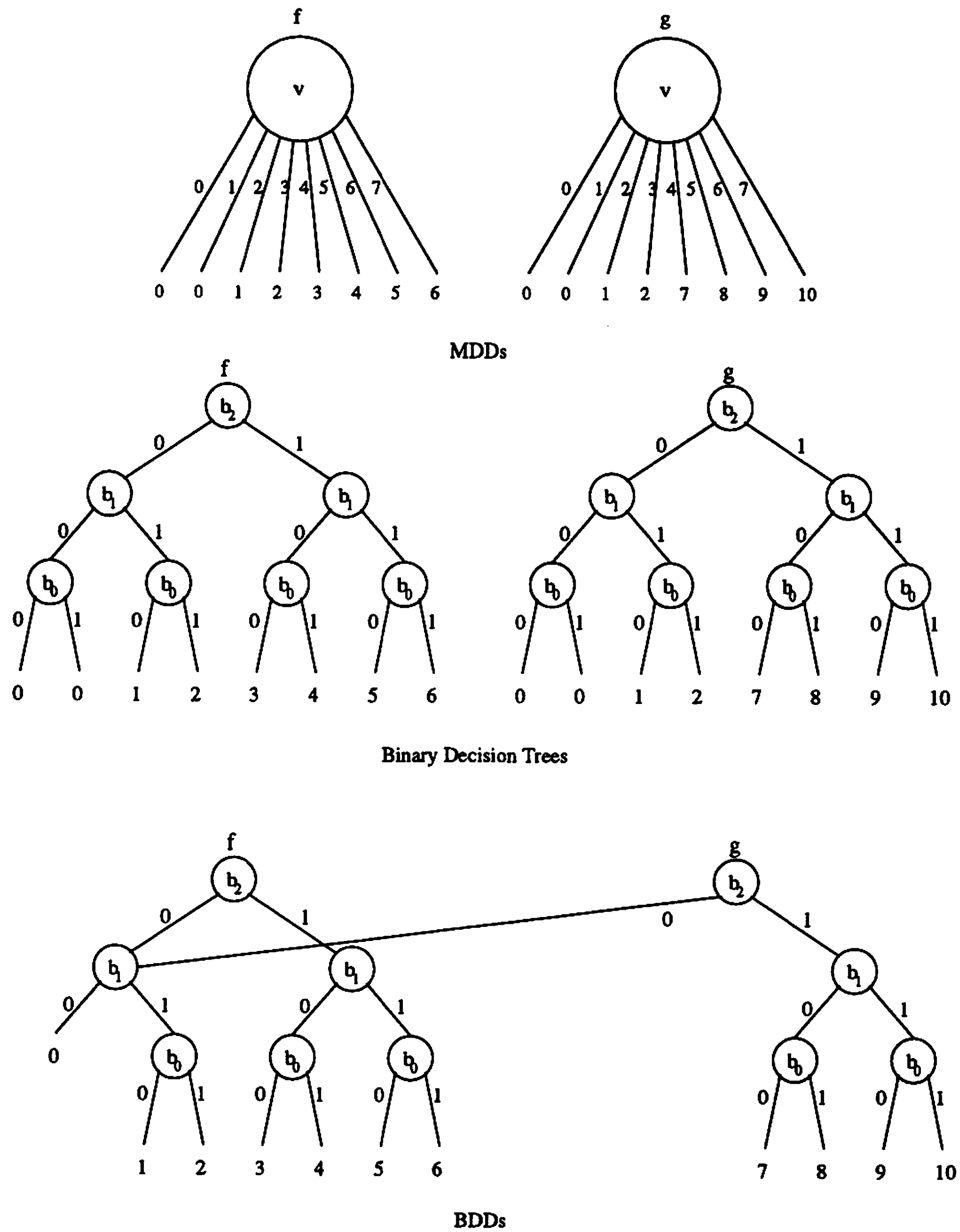


Figure 13: Gain Calculation Example

Input: A compatible C
Output: The gain of C

```
if C = ∅ then
  return 0
end if

gain = 0

/* Get top level sets */
Top = {c | c ∈ C and no d ∈ C contains c}
𝓘 = {I | I ∈ Top and I is an isomorphic set}

/* 𝒥 contains isomorphic sets whose symbolic lists are not subsets
of any symbolic list of any other isomorphic sets.  𝒥' contains
isomorphic sets whose symbolic lists are subsets of some symbolic
lists of some other isomorphic sets.  Symbolic lists that are subsets
of other symbolic lists are removed from 𝒥' */
𝒥 = ∅
𝒥' = ∅
foreach I ∈ 𝓘 do
  found = FALSE
  foreach I^aux ∈ 𝓘 do
    I' = I
    if ∃l ∈ I, l^aux ∈ I^aux Sym(l) ⊂ Sym(l^aux) then
      I' = I'\{l}
      found = TRUE
  if found = TRUE then
    𝒥' = 𝒥' ∪ {I'}
  else
    𝒥 = 𝒥 ∪ {I}

/* Add gains contributed by 𝒮 */
𝒮 = {S | S ∈ Top and S is a sibling set}
for each S ∈ 𝒮 do
  gain = gain + gain(S)
end for

/* Add gains contributed by 𝒥' */
for each I ∈ 𝒥' do
  gain = gain + gain(I)
end for

/* Recursively add gains contributed by 𝒥 */
for each I ∈ 𝒥 do
  C_s = {c | c ∈ C, ∀l ∈ c Sym(l) ⊂ Sym(l^0), l^0 is the 0-th list of I}
  gain = gain + Gain(𝓘_s)
end for
return gain
```

Theorem 5.8 *Given a compatible C,* Gain*(C) computes the gain of C.*

Proof: This is an inductive proof. At step i we have a set $C_i \in C$ of sibling sets $\mathcal{S}_i$ and isomorphic sets $\mathcal{I}_i$. $\mathcal{S}_i$ and $\mathcal{I}_i$ are sets that are contained completely in $\mathcal{S}_{i-1}, \mathcal{S}_{i-2}, \ldots, \mathcal{S}_0$ and $\mathcal{I}_{i-1}, \mathcal{I}_{i-2}, \ldots, \mathcal{I}_0$ and not contained in any other sets in C. At step i, we compute the total gain g_i of C_i (i.e., $\mathcal{S}_i, \mathcal{S}_{i-1}, \ldots, \mathcal{S}_0$ and $\mathcal{I}_i, \mathcal{I}_{i-1}, \ldots, \mathcal{I}_0$). Let $\mathcal{J}_i$ be the set of isomorphic sets of $\mathcal{I}_i$ that do not have any symbolic lists that are subsets of any symbolic lists of any isomorphic sets of $\mathcal{I}_i$. Let $\mathcal{J}_i'$ be the set difference of $\mathcal{I}_i$ and $\mathcal{J}_i$, with the symbolic lists of isomorphic sets that are subsets of those of isomorphic sets in $\mathcal{J}_i$ removed. To illustrate what $\mathcal{J}_i$ and $\mathcal{J}_i'$ represent, we look at the binary decision trees for functions f_0, f_1, and f_2 in Figure 14. In this figure, T_3 and T_4 are isomorphic and T_0, T_1, and T_2 are isomorphic. Let T_i' denote the symbolic list corresponding to the symbols whose codes are represented by subtree T_i. Algorithm 3 generates isomorphic sets $I_0 = \{T_3', T_4'\}$ and $I_1 = \{T_0', T_1', T_2'\}$ among other sets. For this example, $\mathcal{J}_i$ will contain $I_0 = \{T_3', T_4'\}$ and $\mathcal{J}_i'$ will contain $I_1' = \{T_2'\}$. When we apply the BDD reduction rules to this example, the isomorphic subgraphs associated with each isomorphic set in $\mathcal{J}_i'$ will be removed by Rule 2. Hence if l^i is the i-th symbolic list of an $I \in \mathcal{J}_i'$, its gain needs to be updated to $|I| \times (|l^0| - 1)$.

- Case $i = 0$. g_0 is simply equal to the total gain of $\mathcal{S}_0, \mathcal{J}_0$, and $\mathcal{J}_0'$, which is what **Gain**(C) computes if we do not allow its recursion.
- Case $i = k$. Assume that **Gain**(C) computes g_k if we allow the recursion k times.
- Case $i = k + 1$. Since the gain g_k implies the merging of isomorphic subgraphs into one subgraph at recursion k, the additional gain going from step k to step $k + 1$ is the reduction applied to any single isomorphic subgraph. It suffices to consider only the first symbolic list of every isomorphic set in $\mathcal{J}_k$. The reason is that the isomorphic subgraphs corresponding to the isomorphic sets of $\mathcal{J}_k'$ form a subgraph

■

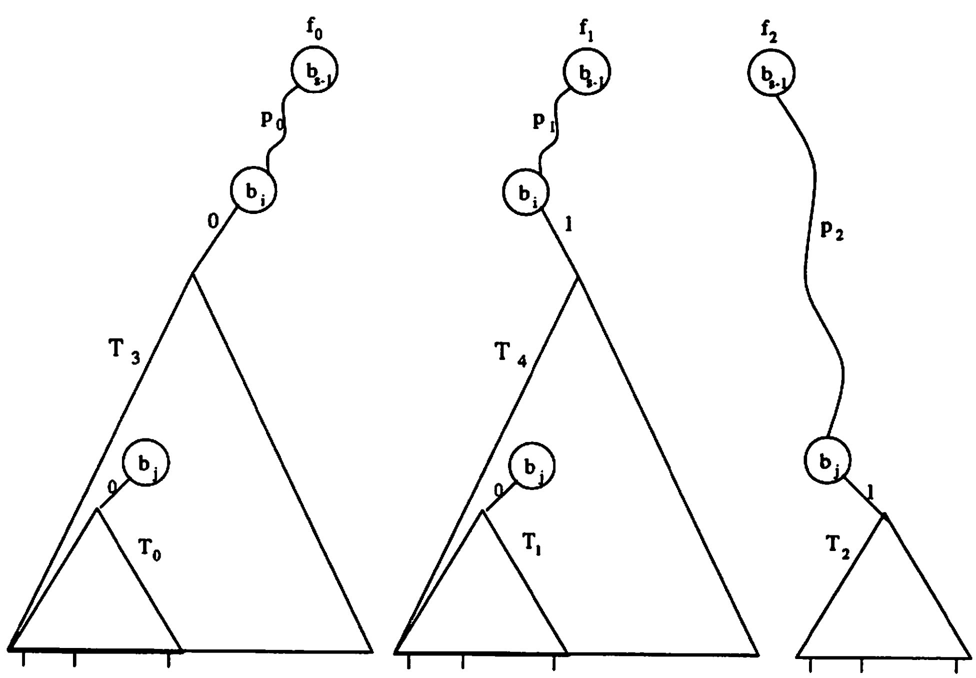


Figure 14: Example for Proving Theorem 5.8

5.2 Maximal Compatibles

Having found all sibling and isomorphic sets, the next task is to find a maximum gain compatible. As shown in the previous section, the gain of a compatible is not proportional to the size of the compatible. In other words, the gain of a compatible may be smaller than the gain of another compatible which contains fewer sets. Luckily, we do not have to enumerate all compatibles to find a maximum gain compatible. A maximal compatible, i.e., a compatible where no set can be added while still maintaining compatibility, always has a larger or the same gain as any proper subset of the compatible. This means that we only need to find all maximal compatibles. A maximum gain compatible is a maximal compatible that has the largest gain among all maximal compatibles.

5.2.1 2-CNF SAT Formulation

We find all maximal compatibles by first building a *compatibility graph*. In the following definition, x denotes either a sibling set or an isomorphic set.

Definition 13 *A compatibility graph $G = (V, E)$ is a labeled undirected graph defined on an instance P of the BDD input encoding problem. There is a vertex x for each set x of P. No other vertices exist. There is an edge $e = (x_1, x_2)$, if and only if x_1 and x_2 are compatible.*

As a consequence of this definition, a compatible of P is a clique in G.

As mentioned above, we need to enumerate all maximal compatibles of P and calculate their gains. Enumerating all maximal compatibles corresponds to finding all maximal cliques of G. The technique we use to find all maximal cliques in G is by first formulating the problem as a 2-CNF SAT formula ϕ and then finding satisfying truth assignments of ϕ. The formula ϕ is created as follows: for each unconnected pair of vertices, x_1 and x_2, we create a clause $(\overline{x_1} \vee \overline{x_2})$. A satisfying truth assignment to ϕ is a set of vertices that do not form a clique. Hence a cube of ϕ is also a set of vertices that do not form a clique. Since ϕ is a unate function, a prime implicant of ϕ contains the minimum number of vertices that do not form a clique. Then the set of vertices that are missing from a prime implicant corresponds to a maximal clique.

In summary, our procedure to find all maximal cliques of G is as follows:

- Formulate the problem into a 2-CNF formula ϕ.
- Pass ϕ to a program, which we call a *CNF expander*, that takes a unate 2-CNF formula and outputs the list of all its prime implicants.
- For each prime implicant, the variables that do not appear in it form a maximal clique.

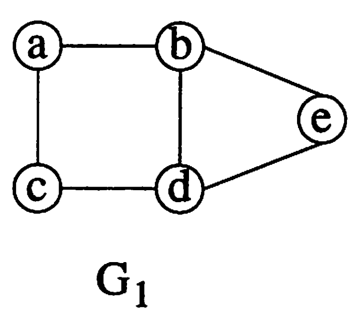


Figure 15: Maximal Clique Problem Example.

For example, consider the graph G_1 in Figure 15. The 2-CNF formula ϕ_1 is

$$\phi_1 = (\overline{a} \vee \overline{d})(\overline{a} \vee \overline{e})(\overline{b} \vee \overline{c})(\overline{c} \vee \overline{e}),$$

where each clause corresponds to a pair of unconnected vertices. The prime implicants of ϕ_1 are $\overline{a}\,\overline{c}, \overline{a}\,\overline{b}\,\overline{e}, \overline{b}\,\overline{d}\,\overline{e}$, and $\overline{c}\,\overline{d}\,\overline{e}$. The maximal cliques of G_1 are bde (corresponding to $\overline{a}\,\overline{c}$), cd (corresponding to $\overline{a}\,\overline{b}\,\overline{e}$, ac (corresponding to $\overline{b}\,\overline{d}\,\overline{e}$), and ab (corresponding to $\overline{c}\,\overline{d}\,\overline{e}$).

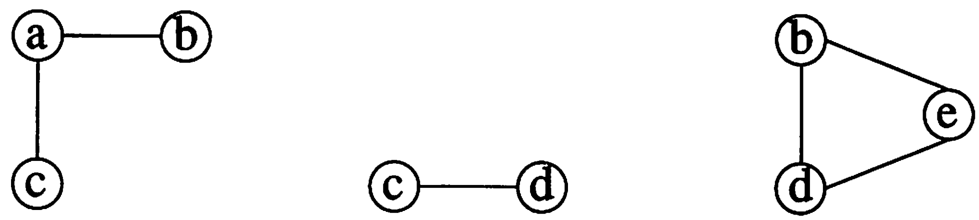
5.2.2 CNF Expander

The CNF expander used here is the one developed by [Vil95]. We explain briefly here how the algorithm works.

The algorithm first simplifies clauses with a common literal, say a, into a single clause with two terms, a and the concatenation of other literals in the original clauses. After all such clauses have been processed, the reduced formula is expanded by multiplying out two clauses at a time. After each multiplication, a single cube containment operation is performed to eliminate non-prime cubes. After all multiplications are done, the result is a list of all prime implicants of the formula. The following example shows how the algorithm expands the formula of Figure 15:

$$\phi_1 = (\overline{a} \vee \overline{d})(\overline{a} \vee \overline{e})(\overline{b} \vee \overline{c})(\overline{c} \vee \overline{e})$$
$$\phi_1 = (\overline{a} \vee \overline{d}\,\overline{e})(\overline{c} \vee \overline{b}\,\overline{e})$$
$$\phi_1 = \overline{a}\,\overline{c} + \overline{a}\,\overline{b}\,\overline{e} + \overline{b}\,\overline{d}\,\overline{e} + \overline{c}\,\overline{d}\,\overline{e}$$

Although this algorithm is linear in the number of prime implicants, the number of clauses that need to be created for a graph with n vertices is proportional to n^2. If n is large and the graph is sparse, this number can be very big. We can reduce the amount of memory that the algorithm needs by partitioning the graph into multiple subgraphs. The idea is to invoke the CNF expander k times. A subgraph of size n_i is passed to the i-th invocation, where each n_i is much smaller than n if the graph is sparse. Then the sum of the squares of all these n_i will be much smaller than n^2.



G_{11} G_{12} G_{13}

Figure 16: Partitioned Graph for Maximal Clique Problem Example.

Gien a graph G, the CNF expander enhanced by partitioning is as follows:

1. Initialize the set of all prime implicant candidates **P** to be an empty set. A prime implicant candidate is an implicant that is either a prime implicant or is covered by a prime implicant of G.
2. Choose a subgraph G_i of G which consists of a smallest degree vertex v and all vertices that are connected to v.
3. Call the CNF expander with G_i as the input.
4. Perform the logic operation AND of all prime implicants of G_i with the complements of the vertices of the original graph G that are not in G_i and include them in **P**. This step is to map the boolean space of G_i into that of the original problem. The mapped terms are the candidate prime implicants.

5. Remove v and all edges that are incident at v from G.
6. If there is more than 1 vertex left, go to step 2.
7. Perform a single cube containment operation on **P**.
8. Return **P**.

It is easy to see that **P** contains all the prime implicants of G at the end of the algorithm.

We illustrate this algorithm on the graph G_1 in Figure 15. We refer to the subgraphs in Figure 16 as we illustrate this example. By choosing a as the smallest degree vertex, the first subgraph we pass to the CNF expander is G_{11}, which consists of a, b and c. The prime implicants of G_{11} are $\bar{b}$ and $\bar{c}$. The candidate prime implicants are $\overline{b\bar{d}e}$ and $\overline{\bar{c}\bar{d}e}$. We then remove vertex a and all its edges from G_1. The smallest degree vertex of the new G_1 is c. Since a has been removed, the only neighbor of c is d. Then the next subgraph G_{12} consists of only c and d. Since G_{12} is a complete graph, the only prime implicant of G_{12} is 1. The only candidate prime implicant of G_{12} is therefore $\overline{a\bar{b}e}$. The only subgraph left after removing c and its edge is G_{13}. Since G_{13} is also a complete graph, the only prime implicant is also 1 and the candidate prime implicant is $\overline{ac}$. Altogether, the set of all candidate prime implicants is $\{\overline{b\bar{d}e}, \overline{\bar{c}\bar{d}e}, \overline{a\bar{b}e}, \overline{ac}\}$, which is also the set of prime implicants of G_1.

As a comparison, the CNF expander without partitioning invokes the CNF expander only once, but with 4 clauses for G_1; whereas the CNF expander with partitioning invokes the CNF expander 3 times, but with a total of 1 clause. Also, the exact algorithm without *permute* calls (which will be explained later) took 165 seconds and 350 seconds of CPU time to find the optimum solutions for the circuits *ellen* and *shiftreg4* respectively using the CNF expander with partitioning. Without partitioning, the executions were timed out after some hours of elapsed time.

5.3 Experimental Results

The experiments were performed on a DEC AlphaServer 8400 5/300 with 2Gb of memory on circuits shown in Table 21. Column 2 of this table lists the number of distinct state transitions regardless of the primary input combinations. Note that *shiftreg3* is a 3-bit shift register and *shiftreg4* is a 4-bit shift register. Column 3 lists the size of the domain or $|D|$.

Beside the exact algorithm, an experiment with a version of the "exact" algorithm, where the two *permute*() calls were removed, was also done. For comparison purposes, the results of both versions of the exact algorithm and the simulated annealing runs are shown in Table 22. CPU times are also included in this table. Circuits whose executions were timed out after one hour of CPU time are not listed. Except for *ellen* and *shiftreg4*, the simulated annealing algorithm finds the optimum solutions.

6 Conclusions

We have presented a simulated annealing algorithm which finds good solutions to the problem of encoding the present state variables of a finite state machine such that its BDD representation has the minimum number of nodes. We applied the simulated annealing algorithm to both the functional and the relational BDD representation of an FSM. We carried forth a systematic set of experiments with simulated annealing, to study how encoding affects the BDD size of finite state machines and we are the first to report such complete data.

We have also presented an exact solution to the BDD input encoding problem. Our exact algorithm characterizes the two BDD reduction rules as combinatorial sets and finds encodable compatible sets with maximum gain to produce the optimum encoding. The simulated annealing algorithm runs much faster than the exact algorithm and gives close-to-optimum results, when the latter are known.

Table 21: Completely Specified FSMs.

Name	Number of Functions	Domain Size
dk15x	7	4
dk17x	4	8
ellen	2	16
ellen.min	2	8
ex6inp	17	8
fstate	19	8
fsync	5	4
maincont	5	16
mc	6	4
ofsync	5	4
pkheader	3	16
scud	48	8
shiftreg4	2	16
shiftreg3	2	8
tav	1	4
tbk	26	32
tbk_m	20	16
virmach	34	4
vmecont	19	32

Table 22: BDD Size for Completely Specified FSMs for Simulated Annealing and the Exact Algorithm (Exact algorithm was run with no permutations for *ellen* and *shiftreg4*).

	Number of BDD Nodes			CPU Time		
Name	SA	Exact w/ permute	Exact w/o permute	SA	Exact w/ permute	Exact w/o permute
dk15x	19	19	19	11.542	0.175	0.148
dk17x	41	41	41	19.673	19.102	4.398
ellen	49	spaceout	46	15.522	spaceout	165.489
ellen.min	21	21	21	4.475	5.129	0.077
fsync	24	24	24	13.122	0.017	0.011
mc	20	20	20	2.704	0.237	0.094
ofsync	24	24	24	13.119	0.019	0.012
shiftreg4	47	spaceout	45	12.574	spaceout	350.147
shiftreg	21	21	21	3.437	4.987	0.074
tav	9	9	9	76.352	0.000	0.002

References

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C(35):677–691, 1986.

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3), September 1992.

[CQC95] G. Cabodi, S. Quer, and P. Camurati. Transforming boolean relations by symbolic encoding. In P. Camurati and P. Eveking, editors, *Proceedings of CHARME '95, Correct Hardware Design and Verification Conference*, volume 987 of *LNCS*, pages 161–170. Springer Verlag, October 1995.

[LMSSV95] L. Lavagno, P. McGeer, A. Saldanha, and A.L. Sangiovanni-Vincentelli. Timed Shannon Circuits: A Power-Efficient Design Style and Synthes is Tool. In *Proceedings of the* 32^{th} *Design Automation Conference*, pages 254–260, June 1995.

[MT96a] Ch. Meinel and T. Theobald. Local encoding transformations for optimizing OBDD-representations of finite state machines. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 404–418, 1996.

[MT96b] Ch. Meinel and T. Theobald. State encodings and OBDD-sizes. Technical Report 96-04, Universität Trier, 1996.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 42–47, 1993.

[Vil95] T. Villa. Encoding problems in logic synthesis. Technical report, UCB/ERL M95/41, 1995.