Selection Graph Coloring: Application to Constrained Encoding

Abstract

This paper aims at introducing the concept of *selection graph* with an application to a non-trivial problem. A selection graph is made of a set of classes, each class having multiple realizations, and of an uncompatibility graph built on these realizations. Coloring a selection graph can be used to solve constrained scheduling, constrained ressource allocation, and dichotomy-based constrained encoding (DBCE). This paper shows how the later problem can be solved with selection graph coloring. This results in the first efficient polynomial-memory exact DBCE algorithm. Experimental results show that it outperforms the best known exact solvers.

1 Introduction

During scheduling, one establishes priority and exclusion constraints between tasks, which can be expressed with an uncompatibility graph. Every task can be implemented with several possible ressources, which have their respective uncompatibility constraints. Each ressource is weighted by its cost. The problem is now to obtain a schedule that takes advantage of the alternative implementations of the tasks to optimize some cost function, namely a combination of the schedule length and the cumulative cost of the ressources chosen to implement the tasks [10]. Constrained ressource allocation offers the same kind of problem.

Selection graph coloring captures the problems described above. More generally, it captures the class of problems expressed with a set of objects, a set of possible implementations for every object, an uncompatibility graph on these implementations, and an additive cost function of the cumulative cost of the implementations.

This paper demonstrates the effectiveness of this approach by formalizing a non-trivial application belonging to the same class of problems, the dichotomy-based constrained encoding problem (DBCE in the sequel), which subsumes other constrained encoding methods, e.g., facet-based constrained encoding (see Section 5.1). This paper explains how to solve selection graph coloring efficiently, which results in the first effective exact DBCE solver running with a polymial memory complexity.

This paper is organized as follows. Section 2 introduces selection graphs. Section 3 defines the DBCE problem, and explains why this problem is difficult. It then expresses DBCE as a selection graph coloring problem, and details some algorithmic aspects. Section 4 proposes an efficient algorithm to color selection graphs. Section 5 discusses experimental results. It shows that the approach presented in this paper beats the best known exact DBCE solvers, and that the heuristic DBCE solver derived from the exact version is competitive with the best known heuristics.

2 Selection Graph

A *selection graph* (G, P) is a graph $G = (V, E)$ with a partition P of a subset of its vertices. An element p of P is called a *class*, and we assume that none is empty. A *choice* consists of removing every but one vertex of some class p from G, and removing class p from P. The vertex which is kept in the graph in place of the vertices of p is the *instance* of class p. The selection graph obtained after performing a set of choices is *derived* from the original selection graph. An *instance* graph is a derived graph where no more choice can be made. Any graph optimization problem is extended to a selection graph by taking the best solution obtained over all its instance graphs.

Selection graph is a generalization of the idea introduced in [2]. A selection graph (G, P) implicitly represents a set of instance graphs that are derived by choosing one instance vertex for every class of P. The number of instance graphs is $\prod_{p \in P} |p|$, thus potentially exponential w.r.t. $|G|$. Intuitively, each class p corresponds to some task or object that can be realized by different implementations, each implementation having its own uncompatibility constraints with all the other implementations.

3 An Example: Dichotomy-Based Constrained Encoding

A k-bit *encoding* of a set of states S is a mapping α from S into $\{0, 1\}^k$. A *dichotomy* is an unordered pair

	$\alpha(s)$		
1	0	0	0
2	1	0	–
3	0	1	1
4	1	1	0

Figure 1: A three-bit encoding.

$\{S_0, S_1\}$ of disjoint subsets of S. An encoding satisfies a dichotomy $\{S_0, S_1\}$ if and only if (iff) there is a bit of the encoding that distinguishes the states of S_0 from S_1's, i.e., this bit has the value 0 for all the states in S_0, and the value 1 for all the states in S_1, or vice versa.

During sequential logic synthesis, one wants to find a minimum-length state encoding that meets some optimality criterions (speed, power, area, testability), as well as some correctness criterions in the case of asynchronous finite state machines (e.g., race-free and hazard-free implementations). Some of these criterions can be expressed with dichotomies [12, 13, 6, 9, 10, 14, 4, 5]. The *dichotomy-based constrained encoding problem* (DBCE) consists of finding a minimum-length encoding that satisfies a given set of dichotomies. Fig. 1 shows a minimum-length encoding of $S = \{1, \ldots, 4\}$ that satifies the dichotomies $\{\{1, 3\}, \{2\}\}$, $\{\{1\}, \{2, 4\}\}$, $\{\{1, 2\}, \{3, 4\}\}$, and $\{\{1, 4\}, \{3\}\}$.

3.1 Previous Work

We discuss the state-of-the-art in solving DBCE exactly, and point out the main bottlenecks of the different approaches proposed in the past.

A *compatible set* is a set of dichotomies that can be simultaneously satisfied by a single bit encoding. One say that two dichotomies are *compatible* iff they form a compatible set. A dichotomy $\{S'_0, S'_1\}$ *subsumes* a dichotomy $\{S_0, S_1\}$ iff $S'_0 \supseteq S_0$ and $S'_1 \supseteq S_1$. Any encoding satisfying a dichotomy v also satisfies the dichotomies v subsumes, thus all subsumed dichotomies can be removed. The first method introduced to solve DBCE is as follows [12]:

(1) Build the set $\mathcal{C}$ of all the maximal compatible sets from the given set of dichotomy D (in $O(\frac{3^{|D|}}{|D|})$).
(2) Find a minimum number of maximal compatible sets that subsume D (NP-complete).

Any DBCE solver that requires the explicit construction of $\mathcal{C}$ is seriously limited, because $|\mathcal{C}|$ can be exponential w.r.t. $|D|$. Moreover, note that compatible sets are necessarily made of pair-wise compatible dichotomies, but the converse is false[1]. This makes computing compatible sets difficult.

A first procedure [12, 14, 10] addressing step (1) consists of computing pair-wise compatible dichotomies, adding to D new dichotomies that express these compatibilities[2], and iterating this process until saturation. This approach is limited to small size problems because of the huge number of dichotomy pairs one needs to examine to build $\mathcal{C}$: it can be as large as $O(|D|^{|D|})$.

An *ordered* dichotomy (S_0, S_1) is an ordered couple of disjoint subsets of S. An encoding satisfies an ordered dichotomy (S_0, S_1) iff there exists a bit of the encoding that has the value 0 for all the state of S_0, and the value 1 for all the states of S_1. The concept of compatibility extended to ordered dichotomies is more manageable, since a set of ordered dichotomies is compatible iff they are pair-wise compatible. One associates two ordered dichotomies with each (unordered) dichotomy, and one obtains $\mathcal{C}$ by first computing the maximal sets of compatible ordered dichotomies, and then converting the laters into unordered dichotomies.

A second procedure [11] addressing step (1) uses this idea to obtain an explicit computation algorithm of $\mathcal{C}$, whose complexity is linear w.r.t. $|\mathcal{C}|$. However, as pointed out earlier, $|\mathcal{C}|$ can be exponential w.r.t. $|D|$.

In [3] two ZBDD based algorithms are presented to solve DBCE. Both algorithms are based on set covering with compatible sets. Although ZBDDs allow to manipulate $\mathcal{C}$ *implicitly*, ZBDDs can still take a non-polynomial memory space. Moreover, these approaches suffer from the irreducibility of the resulting set covering problems.

3.2 DBCE and Selection Graph Coloring

The methods that have been proposed in the past to solve DBCE rely on building (explicitly or implicitly) the set $\mathcal{C}$ of maximal compatible sets. This section formalizes DBCE as a selection graph coloring problem, which does no longer require the explicit notion of compatible sets. Section 4 will explain how to color a selection graph.

Let us define the selection graph (G, P) associated with a set of (un)ordered dichotomies D as follows.

- the set of vertices V is the set of ordered dichotomies obtained from D, i.e., each unordered dichotomy

[1] E.g., $\{\{1, 2\}, \{\}\}$, $\{\{1, 3\}, \{\}\}$, and $\{\{2\}, \{3\}\}$, are pair-wise compatible, but they do not form a compatible set: the first two dichotomies force 2 and 3 to have the same encoding bit, while the third dichotomy requires these two states to have a distinguishing bit encoding.

[2] E.g., the compatibility of $\{\{1\}, \{2, 3\}\}$ and $\{\{1, 4\}, \{2\}\}$ is denoted with the new dichotomy $\{\{1, 4\}, \{2, 3\}\}$.

$\{S_0, S_1\}$ of D is replaced in D with the two ordered dichotomies $v = (S_0, S_1)$ and $v' = (S_1, S_0)$.

- the graph $G = (V, E)$ is the uncompatibility graph of these ordered dichotomies, i.e., there is an edge between two ordered dichotomies iff they are uncompatible. This graph is built using the following property that can be easily checked: two ordered dichotomies (S_0, S_1) and (S'_0, S'_1) are uncompatible iff $(S_0 \cup S'_0) \cap (S_1 \cup S'_1) \neq \emptyset$.

- P is made of all the couples $\{v, v'\}$ of ordered dichotomies that replace the unordered dichotomies of D.

Theorem 1 *The minimum coloring of the selection graph* (G, P) *associated with a set of dichotomies* D *gives the minimum encoding satisfying* D.

Proof. By definition, an independent set of the uncompatibility graph G is a set of pair-wise compatible ordered dichotomies, i.e., a compatible set. Thus any k-coloring of the uncompatibility graph, which partitions V into k independent sets, yields a k-encoding. However, a minimum coloring of the uncompatibility graph does not necessarily produce a minimum encoding. This is because each unordered dichotomy generates two ordered dichotomies, and that *only one* of those needs to be satisfied, i.e., needs to be colored. Thus we have to find the minimum coloring over all the graphs obtained by keeping only one out the two ordered dichotomies produced by every unordered dichotomy, which is precisely the definition of coloring the selection graph associated with D. □

Fig. 2 shows on the left the selection graph G associated with the dichotomies $\{\{2,3\},\{\}\}$, $\{\{1,3\},\{2\}\}$, $\{\{1,2\},\{3,4\}\}$, and $\{\{4\},\{\}\}$. Each class is denoted with a bold box. The middle graph is derived by choosing the left most vertex in the first class. The vertices that have been removed and their incident edges are shown with dotted lines. The instance graph on the right shows an optimum coloring of the original selection graph with 3 colors (while its underlying complete uncompatibilty graph needs 6 colors).

When D is only made of unordered dichotomies, its associated graph is symmetric, as shown in Fig. 2. Because of this symmetry, the reader may think that the minimum coloring of the instance graph resulting from choosing instance classes on the same side (as in Fig. 2) produces the optimum coloring of the selection graph. This is not true. For instance, consider the dichotomies $\{\{1,3\},\{\}\}$, $\{\{2\},\{3\}\}$, and $\{\{\},\{1,2\}\}$. Fig. 3 shows the associated selection graph. The middle graph is the

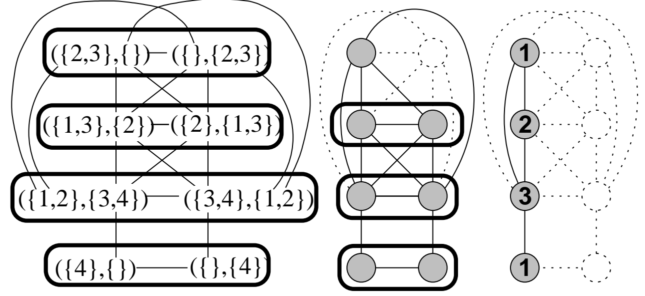

Figure 2: DBCE and selection graph coloring.

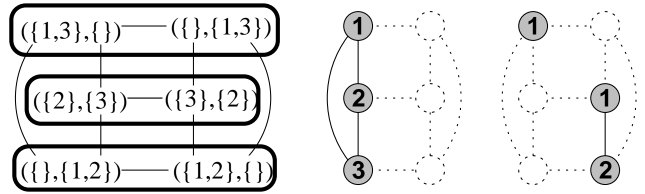

Figure 3: Coloring and symmetry breaking.

instance graph isomorphic to the uncompatible graph of the undordered dichotomies, whose chromatic number is 3. However, the minimum coloring of the selection graph is made of 2 colors only, as shown on the right. It can only be obtained on an instance graph derived with non symmetric choices. The reason is that the ordering of the dichotomies creates new edges that encode the *uncompatibility* of sets made of otherwise *pairwise compatible* unordered dichotomies: in this example, the unordered dichotomies are pairwise compatible, but the three together does not form a compatible set.

3.3 Algorithm

The algorithm solving DBCE is summarized as follows:

(1) Compute a set of irredundant (i.e., there is no subsumed dichotomy), reduced (see Section 3.3.1), ordered dichotomies V from the set of unordered dichotomies D (in $O(|S| \times |D|^2)$).

(2) Build the uncompatibility relation on V to obtain the selection graph (in $O(|S| \times |V|^2)$).

(3) Color the selection graph (NP-complete).

(4) Build an encoding from the coloring (in $O(|V| \times |S|)$).

This algorithm avoids the bottlenecks of the previous methods: it does not need to build the set of maximal

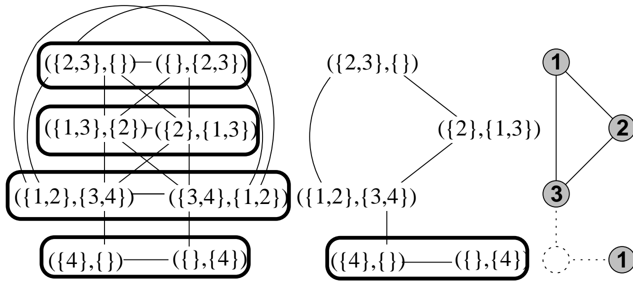

Figure 4: Reduced set of dichotomies.

compatible sets $\mathcal{C}$, whose potential exponential size is an obvious limiting factor. Indeed this is a polynomial size memory algorithm (the dominating factor is the coloring that can be done efficiently with a $O(|D|^3)$ memory complexity, see Section 4), which is not the case for any other proposed methods, including BDD/ZBDD based techniques.

Details of steps (1) and (4) are given below. Step (3) will be addressed in Section 4.

3.3.1 Reduced Set of Ordered Dichotomies

The selection graph associated with D has two ordered dichotomies for every unordered dichotomy of D. However, one can reduce the size of this graph (and as a side-effect, break its symmetry) by generating a reduced set of ordered dichotomies.

Every ordered dichotomy (S_0, S_1) expresses that some bit of the encoding has the value 0 for all states of S_0, and the value 1 for all states of S_1. Since complementing a bit of an encoding still preserves the dichotomy constraints, one can force one state to have an all-zero encoding. Let s be this state. Then one can remove every ordered dichotomies (S_0, S_1) generated from D such that $s \in S_1$. Minimizing the number of resulting ordered dichotomies consists of choosing s as a state that occurs the most in D, which can be done in time linear w.r.t. the size of D.

Fig. 4 shows such a reduction on the same example of Fig. 2. On the left, the complete selection graph. In the middle, the reduced graph obtained by assigning an all-zero encoding to state 2, thus removing every ordered dichotomies (S_0, S_1) such that $2 \in S_1$. On the right, the coloring of the reduced selection graph. Note how the minimum coloring is easier to determine since there is only one class, thus only two instance graphs to examine (both are 3-colorable).

3.3.2 Building an Encoding From the Coloring

Once a coloring is found, one associates one bit per color. A bit has the value 0 (respectively 1) for a state s iff there is an ordered dichotomy (S_0, S_1) colored with this bit's color such that $s \in S_0$ (respectively $s \in S_1$).

Consider the example shown in Fig. 4. Since the ordered dichotomy $(\{2\}, \{1, 3\})$ is colored with color 2, the 2nd bit takes the value 0 for the state 2, and the value 1 for the states 1 and 3. The 3-bit encoding yielded by the 3-coloring is:

	$\alpha(s)$		
1	–	1	0
2	0	0	0
3	0	1	1
4	1	–	1

4 Coloring a Selection Graph

A classical sequential graph coloring algorithm picks a vertex v, colors it with a color non conflicting with v's neighbors, and iterates until all vertices are colored. Backtracking is forced to find better solutions, or when one cannot color a vertex without conflict [1].

One can sequentially color a selection graph by having two possible actions. (1) Perform a choice: pick a class p, choose its instance v, and remove the set of vertices $p - \{v\}$ from the graph. (2) Color a vertex: pick a vertex that does not belong to any class –we shall call it a class-free vertex–, and color it. The performance of this algorithm is very dependent on the action taken at each step. We now discuss the possible strategies and heuristics to decide which action should be carry out at each recursion.

4.1 Choosing a Vertex to Color

A good heuristic to select a vertex to be colored is the DSATUR algorithm [1]. It consists of picking the vertex that has the largest saturation number (i.e., the number of forbidden colors, which are the colors used by its neighbors), and in breaking ties with the largest degree in the uncolored graph. The idea is to choose the vertex that is the most "difficult" to color, and that propagates as many color constraints as possible.

4.2 Performing a choice

Let us assume that we have a function *HardToColor* that estimates the hardness of coloring a vertex v (the greater the value of *HardToColor*(v), the more difficult v is to color). Let (G, P) be a selecting graph, and p be a class of P. Since one can choose any vertices of p as

its instance, we can infer two natural heuristics: (1) the instance of a class p should be the vertex that minimizes *HardToColor* over all vertices of p; (2) the hardness of coloring a class p should be estimated with the hardness of coloring its instance. Let us denote *Instance* the function that implements the first heuristic:

$$\mathit{Instance}(p) \quad = \quad \arg\min_{v \in p} \mathit{HardToColor}(v).$$

A choice consists of choosing the instance of a class. We want to pick the instance of the most difficult class to color, which is:

$$\arg\max_{p \in P} \mathit{HardToColor}(\mathit{Instance}(p))$$

4.3 Strategy

There are two extreme strategies to alternate coloring and choosing class instances:

(1) Strategy "color as late as possible": choose all the class instances first, then color the resulting instance graph.

(2) Strategy "color as soon as possible": whenever a class instance is chosen, it is immediately colored.

Strategy (1) performs poorly for the following reasons. If one selects the "wrong" class instance v for some class p in the early stages of the recursions, one has to exhaust the search for an minimum coloring on a suboptimal derived graph before backtracking and considering another instance of p. Moreover, there is no color constraint information to help the class instance selection.

Strategy (2) performs reasonably well. However it can underperform when classes are far harder to color than class-free vertices

The general strategy, which alternates class instance selection and class-free vertex coloring in any order, opens many alternative heuristics. The simplest of them merely merges both selection heuristics, i.e., it consists of computing the vertex v maximizing *Select*:

$$\mathit{Select}(v) = \begin{cases} \mathit{HardToColor}(v) \text{ if } v \text{ is class free} \\ \mathit{HardToColor}(v) \text{ if } \exists p \in P, v = \mathit{Instance}(p) \\ -\infty \text{ otherwise} \end{cases}$$

If the resulting vertex is class free, we color it, else we select it as the instance of the class it belongs to. We use this heuristics in our implementation, where *HardToColor* is derived from the DSATUR heuristic. In practice, this strategy closely mimics strategy (2), but overcomes its limitation.

4.4 Recursion Pruning

Pruning recursions when coloring selection graphs (G, P) is difficult. Usually, a lower bound on the chromatic number (i.e., the number of colors) is obtained as the size of a clique found in G. The initial clique used as a starting point for the coloring must be made of vertices that are guaranteed to be kept in the graph, thus the clique is made of class-free vertices only. Consequently the clique is very small in practice, and does not help much. Dynamically recomputing a clique after some choices has been performed is costly and still fairly ineffective.

One way to enhance pruning specifically for DBCE is as follows. A partial coloring yields a partial encoding, thus every uncolored dichotomies the partial encoding satisfies can be removed from the selection graph, which prunes useless recursions.

5 Experimental Results

The benchmark consists of MCNC industrial examples representing a wide range of FSMs. We compared the algorithm presented in this paper, that we will call SGC for Selection Graph Coloring, with the best known previous exact DBCE solvers, [11] and [3]. The results are summarized in Table 2, and have been ran on the same workstation (we omitted the smallest examples that are easy to solve). For SGC, the CPU time includes reading the constraints, translating them into a set of irredundant unordered dichotomies, them into a reduced set of ordered dichotomies, building the corresponding selection graph, solving its minimum coloring, and building the minimum constrained state encoding out of it. The characteristics of the reduced selection graphs are also given in the table.

SGC consistently beats the two other methods, and it can solve problems that fail to terminate otherwise. Note that SGC finds the first known exact solution for the long-standing *tbk* example (the minimum solution is 17, while the best known upper bound was 18). When limiting the number of recursions to 5000 backtracks, one obtains a heuristic solver, which finds the optimum solution in all but 4 cases —in these cases it finds an encoding using one extra bit—.

5.1 DBCE and facet-based constrained encoding

Face hypercube embedding is another constrained state encoding framework [11]. A *facet* f is a set of states that are constrained to be encoded in one cube (face) of the hypercube $\{0, 1\}^k$, without having any other state

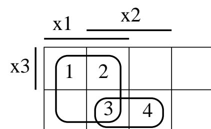

Figure 5: Face hypercube embedding.

intersecting this cube. Fig. 5 shows an optimal face embedding of the two facets $f_1 = \{1, 2, 3\}$ and $f_2 = \{3, 4\}$. The cubes spanned by f_1 and f_2 are 1−− and –10 respectively.

A set of facets can be easily translated to a set of dichotomies: for every facet f, generate the dichotomy $\{f, \{s\}\}$ with $s \in S - f$, which asserts that no other state intersects the face spanned by f; also add the dichotomies $\{\{s\}, \{s'\}\}$ for $s, s' \in S, s \neq s'$, which asserts that every state is distinguishable. However, there is no systematic way to translate a set of dichotomies into a set of facets. Thus DBCE is strictly more expressive than face hypercube embedding (in particular, DBCE can express constrains for asynchronous FSM that cannot be captured with facets). Both DBCE and face hypercube embedding are NP-complete, but DBCE is notoriously harder to solve in practice.

MINSK is the most effective method to solve face hypercube embedding [7] exactly. MINSK cannot solve DBCE, and extending the capacity of MINSK to capture the expressive power of dichotomies would require solving a satisfiability problem at every recursion performed by MINSK, which is impractical [8]. Thus comparing MINSK and SGC should not be to the advantage of SGC, since the later solves problems in a language which is strictly more powerful. However, table 1 shows the results obtained by both methods for some hard face hypercube embedding problems (the CPU time for MINSK is given for a 300 MHz Alpha [7]). There are examples where one method fails to find the optimum encoding while the other does, and conversely. One interesting point is that for some examples ($s1488$, $s1494$, $s298$, *tbk*), MINSK cannot even produce a valid (i.e., non necessarily minimum) encoding in less than 2h. The reason is that MINSK tries to find a solution with a given code length k (initially k is the minimum possible number of bits, i.e., $\lceil \log_2(|\mathbf{S}|) \rceil$), and if it fails, it increments k and iterates. Consequently MINSK works poorly with large value of the code length [8], i.e., large value of the number of colors in the associated selection graph, which SGC can handle nicely.

Example				MINSK	SGC
FSM	$\|\mathbf{S}\|$	$\|\mathbf{F}\|$	**min**/k	(300 MHz)	(167 MHz)
dk16	27	23	5/6	161.45	(7/0.93)
dk512	15	9	4/5	0.06	27.77
donfile	24	24	5/6	48.14	(7/1.45)
ex1	20	8	5/7	0.47	0.05
ex2	19	8	5/6	0.13	1.23
ex4	14	1	4/4	0.00	0.07
keyb	19	18	5/7	1.62	0.23
planet	48	10	6/6	0.40	1.66
s1	20	5	5/5	0.03	3.53
s1488	48	24	6/13?	–	(13/1.04)
s1494	48	24	6/16?	–	(16/0.95)
s298	218	41	8/14?	–	(14/104.23)
s386	13	5	4/6	0.02	1.38
sand	32	5	5/6	0.02	(6/0.28)
scf	121	14	7/7	2.82	(8/4.78)
styr	30	16	5/6	0.29	0.80
tbk	32	73	5/**17**	–	22.75

The CPU times are given in seconds on a 300 MHz Alpha for MINSK, and on a 167 MHZ UltraSparc for SGC.
In case the solver did not finish in less than 2h, we give the best solution it founds and the CPU time it took to find it in parenthesis. "–" means it is unable to obtain a solution in less than 2h.

Table 1: Face hypercube embedding: MINSK and SGC.

6 Conclusion

Selection graphs can be used to formalize problems involving an uncompatibility relation between objects that can have several implementations. For this purpose, this paper showed that coloring a selection graph is an effective paradigm to solve a non-trival application, dichotomy-based constrained encoding (DBCE). It proposed an algorithm to optimally color a selection graph and discussed the different strategies and heuristics that take place during the resolution. Experimental results showed that the resulting algorithm beats the best known exact DBCE solvers.

There are other applications of selection graph coloring, e.g, constrained scheduling and constrained ressource allocation. Selection graph coloring is a difficult problem, but it opens new opportunities to tackle well-known problems, exactly and heuristically. In particular, sophisticated vertex and action selection in the selection graph coloring algorithm could lead to better, more robust, and faster optimization methods.

Finally, the way a "standard" sequential graph coloring algorithm has been extended to color a selection graph can also be applied to other problems, e.g., maximum clique. Solving these other problems on selection graphs may help solving other CAD optimization problems.

References

[1] D. Brélaz, "New Methods to Color Vertices of a Graph", *Comm. of the ACM*, **22**-4, pp. 251–256, 1979.

[2] O. Coudert, "A New Paradigm for Dichotomy-based Constrained Encoding", in Proc. of *Design, Automation and Test in Europe (DATE)*, Paris, France, Feb. 1998.

[3] O. Coudert, C.-J. Richard Shi, "Exact Dichotomy-based Constrained Encoding", in Proc. of *ICCD'96*, Austin TX, Oct. 1996.

[4] S. Devadas, A. R. Newton, "Exact Algorithms for Output Encoding, State Assignment, and Four-level Boolean Minimization", *IEEE Trans. CAD*, **1**-10, pp. 13–27, Jan. 1991.

[5] S. Devadas, A. R. Wang, A. R. Newton, A. L. Sangiovanni-Vincentelli, "Boolean Decomposition of Programmable Logic Arrays", pp. 2.5.1–2.5.5 in *IEEE Custom Int. Cir. Conf.*, 1988.

[6] R. M. Fuhrer, B. Lin, S. M. Nowick, "Symbolic Hazard-free Minimization and Encoding of Asynchronous Finite State Machines", pp. 604–611 in *Proc. of IEEE/ACM Int'l Conf. on CAD*, Nov. 1995.

[7] E. I. Goldberg, T. Villa, R. K. Brayton, A. L. Sangiovanni-Vincentelli, "A Fast and Robust Exact Algorithm for Face Embedding", in Proc. of *ICCAD'97*, pp. 296–303, Nov. 1997.

[8] E. I. Goldberg, personal communication, April 1998.

[9] G. D. Micheli, R. K. Brayton, A. L. Sangiovanni-Vincentelli, "Optimal State Assignment for Finite State Machines", *IEEE Trans. CAD*, **4**-3, pp. 269–285, July 1985.

[10] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[11] A. Saldanha, T. Villa, R. K. Brayton, A. L. Sangiovanni-Vincentelli, "Satisfaction of Input and Output Encoding Constraints", *IEEE Trans. CAD*, **13**-5, pp. 589–602, May 1994.

[12] J. H. Tracey, "Internal State Assignment for Asynchronous Sequential Machines" *IEEE Trans. Elec. Comput.*, pp. 551–560, Aug. 1966.

[13] S. H. Unger, "A Row Assignment for Delay-free Realizations of Flow Tables Without Essential Hazards", *IEEE Trans. Elec. Comput.*, **17**-2, pp. 145–158, Feb. 1968.

[14] S. Yang, M. J. Ciesielski, "Optimum and Suboptimum Algorithms for Input Encoding and its Relationship to Logic Minimization", *IEEE Trans. CAD*, **10**-1, pp. 4–12, Jan. 1991.

Example				Selection graph				CPU		
FSM	**|S|**	**|D|**	**min**/k	**|V|**	**|E|**	**|P|**	**#back**	**[3]**	**[11]**	**SGC**
bbsse	16	60	4/6	75	1082	27	28681	9.50	2.38	1.66
bbtas	6	9	3/3	13	26	4	0	0.01	0.02	0.03
beecount	7	18	3/4	22	167	6	19	0.02	0.03	0.03
cse	16	59	4/5	81	1612	28	425	1.84	15.46	0.07
*dk*14	7	25	3/4	28	321	7	0	0.01	15.43	0.03
*dk*15	4	10	2/3	12	43	2	0	0.01	0.01	0.03
*dk*16	27	368	5/6	632	54730	264	–	–	∗	–
*dk*17	8	31	3/4	46	446	15	5	0.01	0.10	0.01
*dk*27	7	21	2/3	30	178	9	56	0.02	0.04	0.02
*dk*512	15	106	4/5	180	5013	74	112638	–	238.72	27.77
donfile	24	480	5/6	860	136207	380	–	–	∗	–
*ex*1	20	71	5/7	114	1909	43	11	128.63	∗	0.05
*ex*2	19	106	5/6	175	5108	70	7088	–	∗	1.23
*ex*3	10	38	4/5	57	701	20	186	0.05	0.31	0.04
*ex*4	14	91	4/4	169	1872	78	0	–	–	0.07
*ex*5	9	31	4/5	29	408	5	11	0.02	0.08	0.02
*ex*6	8	22	3/4	8	196	0	0	0.02	0.04	0.03
*ex*7	10	36	4/5	40	643	11	12	0.02	0.13	0.04
keyb	19	99	5/7	150	5062	51	441	14.43	125.2	0.23
kirkman	16	50	4/6	80	1057	30	2947	n/a	n/a	0.27
lion	4	6	2/2	3	11	0	0	0.01	0.01	0.01
*lion*9	9	44	4/4	65	1073	21	0	0.01	0.24	0.03
*mark*1	15	77	4/5	117	1780	48	4324	1.00	23.43	0.42
mc	4	6	2/2	9	12	3	0	0.01	0.01	0.01
*modulo*12	12	66	4/4	121	1100	55	0	28679.3	21.05	0.04
opus	10	33	4/4	56	326	23	0	1.59	0.31	0.03
planet	48	767	6/6	1435	79751	668	4722	–	∗	1.66
*s*1	20	131	5/5	240	3678	109	15466	–	–	3.53
*s*1*a*	20	131	5/5	240	3678	109	15466	–	–	3.42
*s*8	5	7	3/3	10	15	3	0	0.01	0.01	0.01
sand	32	363	5/6	693	16928	330	–	–	∗	–
scf	121	5683	7/7	11003	1371017	5320	–	–	∗	–
shiftreg	8	28	3/3	43	375	15	0	0.01	0.09	0.03
sse	16	60	4/6	75	1082	27	28681	19.21	2.34	1.70
styr	30	193	5/6	326	9039	133	1168	–	–	0.80
tav	4	6	2/2	12	24	6	0	0.01	0.01	0.01
tbk	32	994	5/**17**	1695	666442	701	635	–	∗	22.75
*train*11	11	96	4/5	157	4383	61	7814	84.22	5.67	2.20
*train*4	4	8	2/2	10	27	2	0	0.01	0.01	0.01

|S| : #states of the FSM.
|D| : #irredundant dichotomy constraints.
min/k: $\lceil \log_2(|\mathbf{S}|) \rceil$/minimum constrained encoding length.
|V| : #nodes (i.e., #reduced ordered dichotomies).
|E| : #edges (i.e., #uncompatible pairs of ordered dichotomies).
|P| : #classes.
#back: #backtracks performed by SGC.
CPU : all CPU times are in seconds on a 167 MHZ UltraSparc Workstation with 96MB ("–" is more than 2h, "∗" is out of memory).

Table 2: Comparing state-of-the-art exact DBCE solvers.