Constraint Bipartite Vertex Cover
Simpler Exact Algorithms and Implementations

Guoqiang Bai and Henning Fernau

Univ. Trier, FB IV—Abteilung Informatik, 54286 Trier, Germany
baiguoqiang@hotmail.com, fernau@uni-trier.de

Abstract. CONSTRAINT BIPARTITE VERTEX COVER is a graph-theoretical formalization of the spare allocation problem for reconfigurable arrays. We report on an implementation of a parameterized algorithm for this problem. This has led to considerable simplifications of the published, quite sophisticated algorithm. Moreover, we can prove that the mentioned algorithm could be quite efficient in practial situations.

1   Introduction

Problem Definition. In this paper, we are considering the following problem:

An instance of CONSTRAINT BIPARTITE VERTEX COVER (CBVC) is given by a bipartite graph $G = (V_1, V_2, E)$, and the parameter(s), positive integers k_1, k_2. The task is: Is there a *vertex cover* $C \subseteq V_1 \cup V_2$ with $|C \cap V_i| \leq k_i$ for $i = 1, 2$?

This graph-theoretic problem can be easily seen to be equivalent to the following problem, namely via the adjacency matrix of a bipartite graph:

An instance of SPARE ALLOCATION (SAP) is given by a $n \times m$ binary matrix A representing an erroneous chip with $A[r, c] = 1$ iff the chip is faulty at position $[r, c]$, and the parameter(s), positive integers k_1, k_2. The task is: Is there a *reconfiguration strategy* that repairs all faults and uses at most k_1 spare rows and at most k_2 spare columns?

Motivation and Previous Work. Kuo and Fuchs [13] provide a fundamental study of that problem. Put concisely, this "most widely used approach to reconfigurable VLSI" uses spare rows and columns to tolerate failures in rectangular arrays of identical computational elements, which may be as simple as memory cells or as complex as processor units. If a faulty cell is detected, the entire row or column is replaced by a spare one. The tacit (but unrealistic) assumption that spare rows and columns are never faulty can be easily circumvented by a parameter-preserving reduction, as exhibited by Handa and Haruki [11]. For technological reasons (avoiding superfluous redundancy [12] as well as too much expensive laser repair surgery), the number of spare rows and columns is very limited (rarely more than fourty), making this problem a natural candidate for a "fixed parameter approach" [5].

However, due to the $\mathcal{NP}$-hardness of the problem (shown in [13]), no poly-time algorithms can be expected. In [9], it is shown that CONSTRAINT BIPARTITE

VERTEX COVER can be solved slightly faster than $\mathcal{O}^*(1.4^k)$. However, that particular algorithm is derived via a very sophisticated analysis of local situations, quite typical for search-tree algorithms that were developed ten years ago. This means that a naive implementation would have to test all these local cases, which is quite a challenging and error-prone task by the sheer number of cases (more than 30). Since many of these cases are quite special, these cases would show up rarely, and therefore programming errors would be hard to detect. Our approach presented here means nearly a complete re-design of the published algorithm: as it is often the case now with "modern" exact algorithms, most of the burden is taken from the algorithm implementor and shifted on the shoulders of the algorithm analyzer.

Contributions. Our main contribution is to present within Sec. 2 implementable algorithm variants of the algorithm given in [9]; to the most refined variant, basically the same run time analysis applies. These variants have been implemented and tested, which will be described in Sec. 3. We observe that in practical instances (generated according to previously described schemes), the algorithms perform much better than it could be expected from theory. This shows that exact algorithms could be useful even in circumstances where real-time performance is important, as it is the case in industrial processes.

Notations. We need some non-standard notation: If $S = (C_1, C_2)$ is a CBVC solution of $G = (V_1, V_2, E)$, then $(|C_1|, |C_2|)$ is called the *signature* $\sigma(S)$ of that solution. We call a solution (C_1, C_2) *signature-minimal* if there is no solution S' with $\sigma(S') < \sigma(S)$; we will also call $\sigma(S)$ a minimal signature in this case. Here, we compare two vectors of numbers componentwisely. The *signature spectrum* of a CBVC instance (G, k_1, k_2) is the set of all minimal signatures (i_1, i_2) (that obey $(i_1, i_2) \leq (k_1, k_2)$). If the parameter is not given, we ignore the latter restriction. As can be seen, all our algorithms (written as decision algorithms for convenience) can be easily converted into algorithms that produce (solutions to) all minimal signatures.

2 Algorithm Variants

We are going to describe three algorithm variants that will be compared later in Sec. 3.

2.1 The Simplest Algorithm A1

It should be noted here that people developing algorithms for VLSI design actually discovered the $\mathcal{FPT}$ concept in the analysis of their algorithms, coming up with $\mathcal{O}(2^{k_1+k_2}k_1k_2 + (k_1+k_2)|G|)$ algorithms in [10,15]. They observed that "if k_1 and k_2 are small, for instance $\mathcal{O}(\log(|G|))$, then this may be adequate." [10, p. 157]. It was in the context of this problem that Evans [6] basically discovered *Buss' rule* to prove a problem kernel for CONSTRAINT BIPARTITE VERTEX COVER. Kuo and Fuchs called this step quite illustratively the *must-repair-analysis*: Whenever a

row contains more than k_2 faulty elements (or a column contains more than k_1 faulty elements, resp.), then that row (column, resp.) must be exchanged.

Lemma 1. CONSTRAINT BIPARTITE VERTEX COVER *has a problem kernel of size* $2k_1k_2$.

The search-tree part (leading to the $\mathcal{O}^*(2^{k_1+k_2})$) is also easy to explain: any edge must be covered (i.e., any failure must be repaired), and there are two ways to do it. This leads to the simplest algorithm variant A1.

Heuristic improvements. From a heuristic viewpoint, it is always good to branch at vertices of high degree (not just at edges) for vertex cover problems, since in the branch when that vertex is not taken into the cover, all its neighbors must go into the cover. A further very important technique is *early abort*. Namely, observe that for the minimum vertex cover C^* of $G = (V_1, V_2, E)$, we have $|C^*| \leq k_1 + k_2$ for any constraint bipartite cover C with $\sigma(C) \leq (k_1, k_2)$. Since those *overall minimum vertex cover*s can be computed in polynomial time using matching techniques (on bipartite graphs), we can stop computing the search-tree whenever the remaining current parameter budget (k_1, k_2) has dropped (in sum) below the overall minimum vertex cover cardinality. Notice that sometimes also the variant that *requires* a minimum vertex cover as constraint vertex cover is considered, e.g., in [4,18] from a parameterized complexity. However, it is not quite clear from a practical perspective why one should insist on overall cover minimality: to the contrary, repairable arrays should be repaired, irrespectively of whether they yield an overall minimum cover or not.

2.2   Triviality Last: Algorithm A2

There are two simple strategies to improve on quite simplistic search-tree algorithms: either (1) there are simple reduction rules that allow one to deduce that (in our case) branching at high-degree vertices is always possible or (2) one can simply avoid "bad branches" by deferring those branches to a later phase of the algorithm that can be performed in polynomial time. We have called the first strategy *triviality first* and the second one *triviality last* in [8]. Notice that both strategies can be used within a mathematical run time analysis for the search tree heuristic sketched in the previous subsection.

   We will explain how to employ the triviality last principle based on the following observations.

Lemma 2. *Let* $G = (V_1, V_2, E)$ *be a connected undirected bipartite graph with maximum vertex degree 2 and let* $\ell = |E|$ *be the number of edges in* G.

1. *If* G *is a cycle, then for* $\ell' := \ell/2$ *we have the minimal signatures* $(0, \ell')$, $(\ell', 0)$ *as well as* $(2, \ell' - 1), (3, \ell' - 2), \ldots, (\ell' - 1, 2)$ *if* $\ell > 4$.
2. *Let* G *be a path.*
   (a) *If* ℓ *is odd, then for* $\ell' := (\ell + 1)/2$ *we have the minimal signatures* $(0, \ell'), (1, \ell' - 1), \ldots, (\ell' - 1, 1), (\ell', 0)$.

(b) If ℓ is even, then for $\ell' := \ell/2 + 1$ we have the minimal signatures
$(0, \ell' - 1), (2, \ell' - 2), \ldots, (\ell' - 1, 1), (\ell', 0)$ if $|V_1| > |V_2|$ and
$(0, \ell'), (1, \ell' - 1), \ldots, (\ell' - 2, 2), (\ell' - 1, 0)$ if $|V_1| < |V_2|$.

Lemma 3. *If the signature spectra of all components of a graph are known, then the signature spectrum of the overall graph can be computed in polynomial time.*

Algorithm 1. CBVC-TL: a still simple search tree algorithm for CBVC

Input(s): a bipartite graph $G = (V_1, V_2; E)$, positive integers k_1 and k_2
Output(s): YES iff there is a vertex cover $(C_1, C_2) \subseteq V_1 \times V_2$, $|C_1| \leq k_1$ and $|C_2| \leq k_2$

    if $k_1 + k_2 \leq 0$ and $E \neq \emptyset$ **then**
       return NO
    else if $k_1 + k_2 \geq 0$ and $E = \emptyset$ **then**
       return YES
5: **else if** possible **then**
       Choose vertex $x \in V_1 \cup V_2$ such that $\deg(x) \geq 3$.
       if $x \in V_1$ **then**
         $d = (1, 0)$
       else
10:      $d = (0, 1)$
       if CBVC-TL$(G - x, (k_1, k_2) - d)$ **then**
         return YES
       else
         return CBVC-TL$(G - N(x), (k_1, k_2) - \deg(x)((1, 1) - d))$
15: **else**
       {vertex selection not possible $\rightsquigarrow$ maximum degree is 2}
       resolve deterministically according to Lemma 4

Proof. The most straight-forward way to see this is via dynamic programming: If (k_1, k_2) is the parameter bound, then use a $k_1 \times k_2$ table that is originally filled by zeros, except the entry at $(0,0)$ that contains a one. Let c loop from 1 to the number of components plus one. In that loop, for each entry (i, j) of that table that equals c and for each element (r, s) of the signature spectrum, we write $c + 1$ as entry into place $(i + r, j + s)$. Finally, the signature spectrum of the overall graph can be read off as those (i, j) whose table entry contains the number of components plus one. This procedure can be further sped up by noting that there could be at most $k_1 + k_2 + 1$ minimal signatures for any graph; so with some additional bookkeeping one can avoid looping through the whole table of size $k_1 \times k_2$. ∎

Lemma 4. CONSTRAINT BIPARTITE VERTEX COVER *can be solved in time* $\mathcal{O}(k \log k)$ *on forests of cycle and path components with budget* k_1, k_2 *(where* $k = k_1 + k_2$*), i.e., on graphs of maximum degree of two.*

We want to point out that Lemma 4 could be seen by a more efficient algorithm than indicated in the proof of Lemma 3. The strategy is the following one:

—First, solve paths of even length.
—Secondly, solve cycles.
—Finally, solve paths of odd length.

Within each of these three categories of components of a graph of maximum degree two, we basically solve small components first (except for cycles as discussed below).

The intuition behind this strategy is that solving paths of even length by an overall minimum cover is the most challenging task, while it is close to trivial for paths of odd length.

- (A) Namely, from Lemma 2, we can easily deduce that a cover is signature-minimal iff it is overall minimum in the case of paths of odd length. Therefore, we can defer the selection of the specific cover vertices to the very end, and this decision could be then taken in a greedy fashion.
- (B) The only difficulty that can show up with covering cycles is that both parameter budgets k_1 and k_2 might be smaller than the length of that cycle. So, when left with cycle components $C_1, \ldots, C_r$ (of length $\ell_1, \ldots, \ell_r$) and paths of odd length, we first test if there are $x_i \in \{0, 1\}$ with $\sum_{i=1}^{r} x_i \ell_i \in \{k_1, k_2\}$. If $\sum_{i=1}^{r} x_i \ell_i = k_j$ for some $x_i \in \{0, 1\}$, we solve those C_i with $x_i = 1$ by covering them with vertices from V_j. More generally, if $\sum_{i=1}^{r} x_i \ell_i \leq k_1$ and $\sum_{i=1}^{r} (1 - x_i) \ell_i \leq k_2$ for some $x_i \in \{0, 1\}$, we solve those C_i with $x_i = 1$ by covering them with vertices from V_1, and the remaining cycles by vertices from V_2. If such x_i cannot be found but the overall minimum for solving the cycle components is less than $k_1 + k_2$, our general greedy strategy (working from small length cycles onwards) will produce one cycle (and only one) that is not solved matching the overall minimum cover, by covering it both with vertices from V_1 and with vertices from V_2.
- (C) The greedy strategy used at the beginning on paths of even length might also lead to a point that some path cannot be solved matching the overall minimum. In that particular case (*), we will deliberately first empty the critical parameter budget. Similar to (A), one can see that this choice is arbitrary for that particular component: any other feasible minimal choice would have served alike. However, since we are working from smaller to larger components, we will never later see a path of even length that cannot be solved to optimum for the reason that we might have "stolen" the optimum solution through step (*). Moreover, the preference of solving small components first is justified by the fact that this way, the smallest distance from the overall minimum is guaranteed. A further justification for the choice of (*) is that this guarantees that (later on) all cycle components will be solved matching the overall minimum.

2.3   More Sophistication: Algorithm A3

The main part of [9] was struggling with improving the branching on vertices up to degree three. Here, we are going to display a much simpler branching

1. If possible: branch at a vertex of degree four or higher.
2. If the graph is polynomially solvable: do so and terminate.
3. if the graph is 3-regular: branch at an arbitrary vertex of degree three.
   // In the following, let c be a component of the graph that is not polynomially solvable and that is not 3-regular.
4. Let A be the vertex in c with the largest number of attached tails.
4a. If A has three tails, then we branch at A. (Notice that c is no S_3.)
4b. If A has only two tails, then consider the neighbor D of A that is not on a tail. Let E be the first vertex of degree three on the path p starting with A, D (E must exist, since p is no tail) and branch at E, with possibly $E = D$.
4c. If A has only one tail p, then branch at A if p is not a microtail.
   // The only tails in c are microtails attached to vertices A with no other tail.
4d. Let A be a vertex with microtail. Let $B \neq A$ be one of the two vertices of degree three that can be reached from A (with possible intermediate vertices of degree two), preferring the closer one, ties broken arbitrarily. Then, branch at B.
   // This way, all microtails are deleted from c.
   // Let A be a vertex of degree two, with the largest number of neighbors that also have degree two. Let B and C be the two vertices of degree three that can be reached from A in either direction, s.t. B is not farther away from A than C.
5. If C is not neighbor of A, then we branch at B.
6. // Now, the vertices B and C of degree three are neighbors of the degree-2-vertex A.
6a. If B and C have three common neighbors A, D, E, either take B, C or A, D, E together into the cover.
6b. If B and C have two common neighbors A, D of degree two, then branch at B.
6c. If B and C have two common neighbors A, D, with D of degree three, then branch at D.
7. // Now, the degree-3-vertices B and C have only one common neighbor, namely the degree-2-vertex A.
7a. If C has three neighbors of degree two, branch at B, and vice versa.
7b. If B or C has exactly one neighbor E of degree three, branch at E.
8. // Now, B and C possess only one neighbor of degree two, namely A.
   // Let $N(B) = \{B_1, B_2, A\}$ and $N(C) = \{C_1, C_2, A\}$.
8a. If $(N(B_1) \cap N(B_2)) = \{B, B'\}$, then branch by either taking B and B' into the cover or taking B_1, B_2 into the cover. The case $|N(C_1) \cap N(C_2)| > 1$ is symmetric.
8b. If $N(B_1) = \{B, B', B''\}$ with $\deg(B'') = 2$, then branch at B'. (There are three possible symmetric cases to be considered.)
8c. Branch at some $X \in (N(B_1) \cup N(B_2) \cup N(C_1) \cup N(C_2)) \setminus \{B, C\}$.

Fig. 1. List of heuristic priorities

strategy (that was actually implemented) to obtain basically the same run time estimate improvements. We describe the adopted branching strategy in what follows. Here, S_3 denotes a star graph with four vertices, one center connected to the three other vertices (and these are all edges of the graph). Notice that we can easily adapt our polynomial-time algorithms described above to cope with S_3-components, as well. So, we term a graph that contains only components of maximum degree two or S_3-components *polynomially solvable*. We also need two further notions from [9]: a *tail* consists of a degree-3-vertex A, followed by a (possibly empty) sequence of degree-two-vertices, ended by a degree-1-vertex. If

case	branching vector	branching number
1.	$(1, 4)$	1.3803
2.	—	no branching
3.	$(1, 3)$	happens only once per search tree path
$4a. - 4c.$	$(2, 3)$	1.3248
$4d. - 5.$	$(3, 3, 4)$	1.3954
$6a.$	$(2, 3)$	1.3248
$6b.$	$(3, 3, 4)$	1.3954
$6c.; 7b.$	$(3, 4, 6, 6, 7)$	1.3954
$7a.$	$(3, 4, 6, 6, 8)$	1.3867
$8a.$	$(2, 4, 5)$	1.3803
$8b.$	$(4, 5, 7, 9, 7, 6, 7, 9)$	1.3905
$8c.$	$(4, 5, 7, 7, 8, 6, 6, 7)$	1.4154

Fig. 2. Branching vectors and numbers for different heuristic priorities

A is neighbor of a degree-1-vertex, we speak of a *micro-tail*. In Fig. 1, we give a list of priorities according to which branching should be done.

Notice that there are further variations of the algorithm that are easily at hand. For example, one can observe that step 4a. can be avoided, since then the component c is a tree for which a list of all minimal signatures can be obtained by dynamic programming. However, this does not affect our worst-case running time analysis that is sketched in the following.

Some Further Comments on the Run-Time Analysis
4d: Recall that there is a micro-tail A' attached to A. The worst case is when $B \in N(A)$, and when $C \in N(A)$ is also of degree three. If B is taken into the cover, then in the next recursive call of the procedure, in the worst case C would be selected for branching, since a (non micro-)tail is attached to C, giving it the highest heuristic priority. If C is put into the cover, then the edge $A'A$ must be covered by an additional vertex. Hence, we obtain the claimed branching vector.
5: In the worst case, $B \in N(A)$ and $C \in N(N(A))$. Since the graph is bipartite, $C \neq B$. If B goes into the cover, then we obtain a (non micro-)tail at C as in case 4d. The case when $B = C$, i.e., B and C are (at least) at distance four, is even better, yielding a branching vector of at least $(2, 3)$.
6: Here, we consider the remaining cases that the degree-2-vertex A is part of a 4-cycle. Notice that the analysis of small cycles was one of the cornerstones of the analysis in [9].
6a: If one out of A, D, E is not going into the cover, then B and C must go there. Conversely, if B or C does not go into the cover, then all of A, D, E are there.
6b: If B goes into the cover, we produce a situation with C having two microtails (case 4b).
6c: There are three subcases to be considered for the run-time analysis, depending on $j = |N(N(C) \setminus \{A, D\}) \cap N(N(B) \setminus \{A, D\})| \in \{0, 1, 2\}$. Notice that $N(B) \cap N(C) = \{A, D\}$, since otherwise case 6 would have applied. Details of the tedious but straight-forward analysis are omitted for reasons of space.

7: Observe that now $|(N(B) \cup N(C)) \setminus \{A\}| = 4$; otherwise, A is part of a 4-cycle or B and C do not have degree three; all these cases were treated above.

7a: If B is taken into the cover, we find a microtail at C. Since both neighbors of C are of degree two, we will branch at their neighbors C_1 and C_2 (not equal to C) at worst. Due to case 6b, we can assume that $N(C_1) \cap N(C_2) = \emptyset$. Taking the branches at C_1 and C_2 alone, we arrive at a branching vector of $(3, 5, 5, 7)$ in the case when B goes into the cover. Altogether, we get a branching vector of $(3, 4, 6, 6, 8)$ which yields a branching number of 1.3867.

7b: If E goes into the cover, we produce a chain of three consecutive degree-2 vertices. Assuming that we have no cycles of length four or six, such a chain can be resolved through a branching vector of $(3, 5, 5, 6)$. Altogether, we arrive at a branching vector of $(3, 4, 6, 6, 7)$, yielding a branching number of 1.3956.

8a: Notice that A, B form a tail after branching at B_1 and B_2. So, the analysis from [9, Table 15] applies.

8b: If B' is taken into the cover, this will be followed by branching at the hitherto unnamed neighbor $\hat{B}$ of B'' by priority 5. We consider the cases that the neighbors of B' or B' and the neighbors of $\hat{B}$ go into the cover. In both cases, we have the possibility to isolate small components (by observing tails) with branching at C and at B_2. Overall, this gives a branching vector of $(4, 5, 7, 9, 7, 6, 7, 9)$ and hence the claimed branching number.

8c: Consider $B' \in N(B_1)$ selected for branching. If B' goes into the cover, then B is neighbor of two vertices of degree two, so that B_2 will be selected for branching, following the analysis of case 7b. If $N(B')$ is put into the cover, A and B are two neighbored vertices of degree two; so, priority 5 applies. Combining the branching vectors yields the claim. Notice that this is by far the worst case branching. The analysis from [9] shows that this particular case can be improved, basically by consequently branching at all vertices from $(N(B_1) \cup N(B_2) \cup N(C_1) \cup N(C_2)) \setminus \{B, C\}$ in parallel. To actually and fully mimic the case distinctions from [9], cycles of length 6 should be treated in a separate way. However, as it turned out, this case (in fact, all subcases of case 8) showed up very rarely in the experiments, so that we could safely omit it. More precisely, less than 0.001% of all branches were due to these "isolated vertices of degree two." So, our implementation is deliberately omitting some of the details from [9] without sacrificing speed in practice. However, it would need only three special cases to be implemented to completely cover case 8c. according to the analysis from [9], namely those depicted in Tables 6, 22 and 23 in [9].

3   The Tests

Blough [2,3] discussed how to model failures in memory arrays. He suggested the so-called center-satellite model, based on earlier work of Meyer and Pradhan [16]. In that model, it is assumed that memory cells could spontaneously and independently fail with a certain probability p_1. However, once a cell failed, its neighbors also fail with a certain probability p_2, and this affects a whole region of radius r around a central element. So, we are dealing with compound
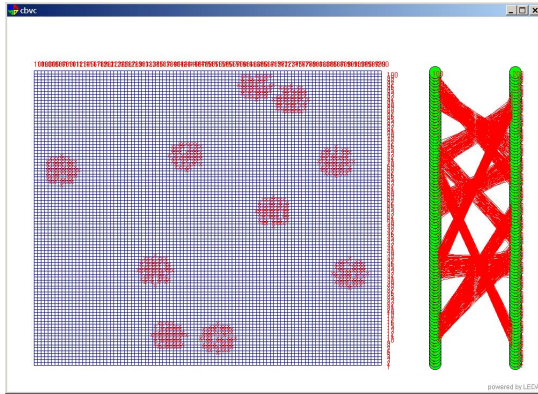
Fig. 3. The graphical interface of the implementation; to the right, the bipartite graph model is shown

probabilities. This leads to a two-stage model to simulate such kind of failures: in a first phase, with probability p_1, memory cells are assumed to be faulty; in a second phase, within a neighborhood of radius r, cells are assumed to be faulty with probability p_2. Fig. 3 shows a failure pattern obtained in this way.

Notice that we assume both underlying distributions to be uniform. This differs a bit from the original model of Meyer and Pradhan [16] who assumed a Poisson distribution for the satellites; however, since we are focussing on rather small radii in our experimental studies, this distinction is not essential. Also Blough and Pelc adopted this simpler approach. However, we deviate from the approach of Blough and Pelc insofar as we are considering cycles around the center with the usual meaning, while they considered cycles in the Manhattan metric, i.e., square-shaped failure regions.

All tests were run on a PC with Athlon XP 2000+ processor (2 GHz with 512 KB Cache) and with 256 MB main memory. We used Microsoft Windows XP Professional, Visual Studio 2003 and LEDA 3.0. All the running times were obtained by testing 100 independently generated instances within a given setup as described in the first columns of the tables, describing the dimensions of the array, the number of spare lines, the chosen probabilities and the chosen radius r (for the satellites). The run times are given in seconds per instance (on average). Fig. 4 tries to simulate the settings described by Blough and Pelc in [3]. We list the measurements separately for the case that a solution was found (or not).

At first glance, the corresponding figures seem to be surprising, since the running times for finding a solution are consistently larger than the ones for rejecting the instance. This phenomenon is partly explained when looking at the average number of branchings (# br) encountered. Obviously, the early abort method (based on maximum matching) allows to early reject most instances nearly without branching in polynomial time. We validated this hypothesis by testing our algorithms without that heuristic, which led to a tremendous slow-down, in particular in the case when no solution was to be found. We also separately list the

m=n	k₁=k₂	p₁	p₂	r	success (%)	without solution				with solution			
						Alg 2	# br	Alg 3	# br	Alg 2	# br	Alg 3	# br
1024	32	0.000007	0.8	5	29	0.1003	12	0.1225	12	0.4440	99	0.4416	98
1024	32	0.000004	0.8	9	11	0.1341	6	0.1337	6	0.2184	32	0.2212	32
1024	32	0.000002	0.8	15	100	-------- ---	---	---------	---	0.0663	3	0.0660	3
1024	36	0.000003	0.8	15	8	0.0402	0	0.0407	0	0.0475	0	0.0488	0
1024	36	0.000005	0.7	9	6	0.0445	1	0.0457	1	0.3705	37	0.3255	37
2048	64	0.000003	0.7	7	3	0.0441	0	0.0425	0	5.2006	574	5.0236	573
2048	64	0.000002	0.5	9	10	0.0345	0	0.0322	0	12.6780	1875	10.6960	1875
4096	128	0.000001	0.7	7	30	0.0714	0	0.0728	0	11.2197	576	11.216	574

Fig. 4. The run times of Blough's setup

m=n	k₁=k₂	p₁	p₂	success (%)	without solution				with solution			
					Alg 2	# br.	Alg 3	# br	Alg 2	# br.	Alg 3	# br
4096	32	0.0000025	0.5	39	0.0213	0	0.0210	0	0.0562	13	0.0526	11
512	64	0.00037	0.5	59	0.0144	0	0.0126	0	0.1247	35	0.1168	30
1024	64	0.000085	0.5	47	0.0181	0	0.0209	0	0.1746	33	0.1838	29
2048	64	0.00002	0.5	78	0.0231	0	0.0254	0	0.1934	33	0.1823	27
4096	64	0.000005	0.5	48	0.0344	0	0.0348	0	0.2010	33	0.1888	24
1024	128	0.00018	0.5	40	0.0235	0	0.0233	0	0.3480	52	0.3020	45
2048	128	0.000042	0.5	52	0.03652	0	0.03502	0	0.7441	78	0.6911	64
4096	128	0.00001	0.5	66	0.0642	0	0.0530	0	0.7803	73	0.7530	57
6144	128	0.0000046	0.5	18	0.0774	0	0.0767	0	0.8389	76	0.8056	54
8192	128	0.0000027	0.4	53	0.0566	0	0.0594	0	0.4752	56	0.3816	36
2048	256	0.0001	0.5	90	0.0600	0	0.0500	0	2.3103	160	1.9757	132
4096	256	0.000021	0.5	80	0.0550	0	0.0550	0	2.6198	154	2.0930	126
6144	256	0.000009	0.5	78	0.0733	0	0.0770	0	2.3590	161	1.8155	121
8192	256	0.0000055	0.4	79	0.0800	0	0.0768	0	1.8086	119	1.4749	82
4096	512	0.000052	0.4	20	0.1101	0	0.1188	0	10.115	335	8.2220	267
6144	512	0.000022	0.4	20	0.1391	0	0.1452	0	9.6835	296	7.7065	224
8192	512	0.0000117	0.4	60	0.1277	0	0.1227	0	8.4538	274	6.6630	205
9216	512	0.0000092	0.4	20	0.1490	0	0.1440	0	8.0715	263	6.7795	198
10240	512	0.0000074	0.4	70	0.1436	0	0.1466	0	8.0216	255	6.0901	183

Fig. 5. The run times for larger numbers

running times for the (quite sophisticated) Alg. 3 in comparison with the much simpler Alg. 2. In practice, it does not necessarily pay off to invest much more time in implementing the more complicated algorithm.

In Fig. 5, we take a more radical approach: while we always restricted the radius of the defective region to one, we explored quite large parameter values. Even with values as high as $k_1 = k_2 = 512$, which gives an astronomic constant of about 2^{500} in our run-time worst case estimates, we still obtain (non-)solutions in only a few seconds. In this sense, our algorithm displays quite a robust behaviour.

It is also seen that the savings of branches of the more complicated Alg.3 are more visible in more complicated instances.

In fact, we also tested on usual random graph models and found similar observations as those reported for Blough's model, see [1]. In another set of experiments (again explicitly reported in [1]), we observed that good run times seem to depend on the fact that, in the experiments displayed so far, $k_1 = k_2$: whenever k_1 and k_2 are much different, the early abort method strikes only occasionally, and therefore the running times do considerably increase.

We can read our experimental results also as a validation of earlier probability-theoretic results indicating that it is unlikely to find hard instances for CBVC under various probability models, including the center-satellite model described above [3,17]. This also explains why the probabilities p_1 and p_2 listed in our figures look quite special: indeed they were searched for in order to display any non-trivial behaviour of our algorithms.

4   Conclusions: Lessons Learned and Future Work

We have shown that $\mathcal{FPT}$ methodology is quite useful at various stages when designing algorithms for $\mathcal{NP}$-hard problems:

- One could validate heuristics against known optimal solutions; this sort of application is useful even when the exact algorithm turns out to be too slow for actual applications within the industrial process at hand;
- At least in some circumstances, one could actually use those algorithms, even when dealing with a (pipelined, but still real-time) industrial application (as in chip manufacturing processes in our concrete application), at least with an additional time limit that might sometimes stop the search tree and output the current (sub-optimum) solution.

In particular, modern exact algorithms are very neat to implement since their overall structure tends to be quite simple: in our case, we could first implement Algorithm A1 as search tree backbone. A1 can be safely implemented in two weeks by one programmer. The integration of the polynomial phase, leading to A2, may take another week including validation. The details of the heuristic priority list may take more time than previously invested, but tests could and should always accompany this phase to see if the special cases considered with those priorities will actually occur. This consideration brought us to the decision to omit some of the special cases (that should have been implemented to fully match the analysis given in [9]), because those cases will not show up very often.

It remains as future work to compare our approach with existing published heuristics and also with some other exact approaches, a recent example being [14]. Even more interesting would be to test our algorithms on "real data", not only on simulated data. Since this particular problem is connected with many production secrets, these real data are not available.

Furthermore, there are alternative yield enhancement strategies employed in modern chip fabrication processes, as sketched in [7]. As detailed in [1], some

of these enhancements can actually be solved with (a slight variant of) the algorithms described in this paper. However, others seem to be more costly within the parameterized algorithm framework. So, the development of efficient parameterized algorithms for these variants is a further challenge for the future.

References

1. Bai, G.: Ein eingeschränktes Knotenüberdeckungsproblem in bipartiten Graphen. Diplomarbeit, FB IV, Informatik, Universität Trier, Germany (2007)
2. Blough, D.M.: On the reconfiguration of memory arrays containing clustered faults. In: Fault Tolerant Computing, pp. 444–451. IEEE Press, Los Alamitos (1991)
3. Blough, D.M., Pelc, A.: A clustered failure model for the memory array reconfiguration problem. IEEE Transactions on Computers 42(5), 518–528 (1993)
4. Chen, J., Kanj, I.A.: Constrained minimum vertex cover in bipartite graphs: complexity and parameterized algorithmics. Journal of Computer and System Sciences 67, 833–847 (2003)
5. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999)
6. Evans, R.C.: Testing repairable RAMs and mostly good memories. In: Proceedings of the IEEE Int'l Test Conf., pp. 49–55 (1981)
7. Fernau, H.: On parameterized enumeration. In: Ibarra, O.H., Zhang, L. (eds.) COCOON 2002. LNCS, vol. 2383, pp. 564–573. Springer, Heidelberg (2002)
8. Fernau, H.: Parameterized Algorithmics: A Graph-Theoretic Approach, Habilitationsschrift, Universität Tübingen, Germany (2005)
9. Fernau, H., Niedermeier, R.: An efficient exact algorithm for constraint bipartite vertex cover. Journal of Algorithms 38(2), 374–410 (2001)
10. Haddad, R.W., Dahbura, A.T., Sharma, A.B.: Increased throughput for the testing and repair of RAMs with redundancy. IEEE Transactions on Computers 40(2), 154–166 (1991)
11. Handa, K., Haruki, K.: A reconfiguration algorithm for memory arrays containing faulty spares. IEICE Trans. Fundamentals E83-A(6), 1123–1130 (2000)
12. Koren, I., Pradhan, D.K.: Modeling the effect of redundancy on yield and performance of VLSI systems. IEEE Transactions on Computers 36(3), 344–355 (1987)
13. Kuo, S.-Y., Fuchs, W.K.: Efficient spare allocation for reconfigurable arrays. IEEE Design and Test 4, 24–31 (1987)
14. Lin, H.-Y., Yeh, F.-M., Kuo, S.-Y.: An efficient algorithm for spare allocation problems. IEEE Transactions on Reliability 55(2), 369–378 (2006)
15. Lombardi, F., Huang, W.K.: Approaches to the repair of VLSI/WSI PRAMs by row/column deletion. In: International Symposium on Fault-Tolerant Computing (FTCS 1988), pp. 342–347. IEEE Press, Los Alamitos (1988)
16. Meyer, F.J., Pradhan, D.K.: Modeling defect spatial distribution. IEEE Transactions on Computers 38(4), 538–546 (1989)
17. Shi, W., Fuchs, W.K.: Probabilistic analysis and algorithms for reconfiguration of memory arrays. IEEE Transactions on Computer-Aided Design 11(9), 1153–1160 (1992)
18. Wang, J., Xu, X., Liu, Y.: An Exact Algorithm Based on Chain Implication for the Min-CVCB Problem. In: Dress, A.W.M., Xu, Y., Zhu, B. (eds.) COCOA. LNCS, vol. 4616, pp. 343–353. Springer, Heidelberg (2007)