Proc. 4th WEA-05, Vol. 3503 in LNCS, pp. 240-252, Springer, 2005

Algorithm Engineering for Optimal Graph Bipartization

Falk Hüffner*

Institut für Informatik, Friedrich-Schiller-Universität Jena, Ernst-Abbe-Platz 2, D-07743 Jena, hueffner@minet.uni-jena.de.

Abstract. We examine exact algorithms for the NP-complete Graph Bipartization problem that asks for a minimum set of vertices to delete from a graph to make it bipartite. Based on the "iterative compression" method recently introduced by Reed, Smith, and Vetta, we present new algorithms and experimental results. The worst-case time complexity is improved from $O(3^k \cdot kmn)$ to $O(3^k \cdot mn)$, where n is the number of vertices, m is the number of edges, and k is the number of vertices to delete. Our best algorithm can solve all problems from a testbed from computational biology within minutes, whereas established methods are only able to solve about half of the problems within reasonable time.

1 Introduction

There has recently been a much increased interest in exact algorithms for NP-hard problems [23]. All of these exact algorithms have exponential run time, which at first glance seems to make them impractical. This conception has been challenged by the view of *parameterized complexity* [6]. The idea is to accept the seemingly inevitable combinatorial explosion, but to confine it to one aspect of the problem, the *parameter*. If for relevant inputs this parameter remains small, then even large problems can be solved efficiently. Problems for which this "confining" is possible are called *fixed-parameter tractable*.

The problem we focus on here is Graph Bipartization, also known as Maximum Bipartite Subgraph or Odd Cycle Transversal. It is NP-complete [13] and MaxSNP-hard [19]; the best known polynomial-time approximation is by a logarithmic factor [9]. It has numerous applications, for example in VLSI design [1, 12], computational biology [21, 18], and register allocation [24].

In a recent breakthrough paper, solving a more than five years open question [14], Reed, Smith, and Vetta [20] proved that the Graph Bipartization problem on a graph with n vertices and m edges is solvable in $O(4^k \cdot kmn)$ time, where k is the number of vertices to delete. The basic idea is to construct size-k solutions from already known size-$(k+1)$ solutions, the so-called *iterative compression*. Their algorithm is of high practical interest for several reasons: the given fixed-parameter complexity promises small run times for small parameter

* Supported by the Deutsche Forschungsgemeinschaft, Emmy Noether research group PIAF (fixed-parameter algorithms), NI 369/4.

values; no intricate algorithmic concepts with extensive implementation requirements or large hidden runtime costs are used as building blocks; and being able to "optimize" given solutions, it can be combined with known and new heuristics.

In this work we demonstrate by experiments that iterative compression is in fact a worthwhile alternative for solving Graph Bipartization in practice. Thereby, we also shed more light on the potential of iterative compression, which has already found applications in other areas as well [3, 4, 11, 16]. The structure of this work is as follows: In Sect. 3 we give a top-down presentation of the Reed-Smith-Vetta algorithm with the goal of making this novel algorithm technique accessible to a broader audience. Moreover, it prepares the ground for several algorithmic improvements in Sect. 4. In Sect. 5 we present experimental results with real-world data (Sect. 5.1), synthetic application data (Sect. 5.2), and random graphs (Sect. 5.3).

2 Preliminaries

By default, we consider only undirected graphs $G = (V, E)$ without self-loops, where $n := |V|$ and $m := |E|$. We use $G[V']$ to denote the subgraph of G induced by the vertices $V' \subseteq V$. For a set of vertices $V' \subseteq V$, we write $G \setminus V'$ for the graph $G[V \setminus V']$. With a *side* of a bipartite graph G, we mean one of the two classes of an arbitrary but fixed two-coloring of G. A *vertex cut* between two disjoint vertex sets in a graph is a set of vertices whose removal disconnects these two sets in the graph.

Definition 1 (Graph Bipartization). *Given an undirected graph $G = (V, E)$ and a nonnegative integer k. Does G have an* odd cycle cover *C of size at most k, that is, is there a subset $C \subseteq V$ of vertices with $|C| \leq k$ such that each odd cycle in G contains at least one vertex from C? Note that the removal of all vertices in C from G results in a bipartite graph.*

We investigate Graph Bipartization in the context of parameterized complexity [6] (see [5, 7, 8, 17] for recent surveys). A parameterized problem is called *fixed-parameter tractable* if it can be solved in $f(k) \cdot n^{O(1)}$ time, where f is a function solely depending on the parameter k, not on the input size n.

For comparison, we examined two alternative implementations: one by Wernicke based on Branch-and-Bound [22], and one based on the following simple integer linear program (ILP):

$$
\begin{aligned}
& C_1, \ldots, C_n, s_1, \ldots, s_n : \text{binary variables} \\
& \text{minimize} \textstyle\sum_{i=1}^{n} C_i \\
& \text{s. t. } \forall \{v, w\} \in E : s_v + s_w + (C_v + C_w) \geq 1 \\
& \qquad \forall \{v, w\} \in E : s_v + s_w - (C_v + C_w) \leq 1
\end{aligned}
$$

The ILP performs surprisingly well; when solved by GNU GLPK [15], it consistently outperforms the highly problem-specific Branch-and-Bound approach by Wernicke on our test data, sometimes by several orders of magnitude. Therefore, we use it as the main comparison point for the performance of our algorithms.

3 A Top-Down Presentation of the Reed-Smith-Vetta Algorithm

In this section we present in detail the algorithm for Graph Bipartization as described by Reed, Smith, and Vetta [20]. While they focus on the correctness proof and describe the algorithm only implicitly, we give a top-down description of the algorithm while arguing for its correctness, thereby hopefully making the result of Reed et al. more accessible.

The global structure is illustrated by the function Odd-Cycle-Cover. It takes as input an arbitrary graph and returns a minimum odd cycle cover.

```
Odd-Cycle-Cover(G = (V, E))
1  V' ← ∅
2  C ← ∅
3  for each v in V
4      do V' ← V' ∪ {v}
5         C ← Compress-OCC(G[V'], C ∪ {v})
6  return C
```

The routine Compress-OCC takes a graph G and an odd cycle cover C for G, and returns a smaller odd cycle cover for G if there is one; otherwise, it returns C unchanged. Therefore, it is a loop invariant that C is a minimum odd cycle cover for $G[V']$, and since eventually $V' = V$, we obtain an optimal solution for G.

It remains to implement Compress-OCC. The idea is to use an auxiliary graph $H(G, C)$ constructed from $G = (V, E)$ and C as follows (see Fig. 1 (a) and (b)):

- Remove the vertices in C from G and determine the sides of the remaining bipartite graph (in Fig. 1 (a), one side comprises $\{b, d\}$ and the other $\{e, f, h\}$).
- For each $c \in C$, add a vertex c_1 to one side and another vertex c_2 to the other side.
- For each edge $\{v, c\} \in E$ with $v \notin C$ and $c \in C$, connect v to that vertex from c_1 and c_2 that is on the other side (see the bold lines in Fig. 1 (b)).
- For each edge $\{c, d\} \in E$ with both $c, d \in C$, arbitrarily connect either c_1 and d_2 or c_2 and d_1 (for example in Fig. 1, we chose $\{g_1, c_2\}$).

The crucial property of the resulting graph H is that every odd cycle in G that contains a vertex $c \in C$ implies a path $(c_1, \ldots, c_2)$ in H. This means that all odd cycles in G can be found as such paths in H, since the vertices in C touch all odd cycles. For example, the triangle d, c, h in G (Fig. 1 (a)) can be found as path (c_1, h, d, c_2) in $H(G, C)$ (Fig. 1 (b)).

Therefore, if we could find a set C' of vertices whose removal disconnects for each $c \in C$ the two vertices c_1 and c_2 in H, then C' is an odd cycle cover for G. Unfortunately, solving this multi-cut problem is still NP-complete. Consider, however, a partition of the vertices $\bigcup_{c \in C} \{c_1, c_2\}$ such that for all $c \in C$ the two

copies c_1 and c_2 are in different classes (called a *valid partition* for C). We can find a vertex cut between the two classes of a valid partition in polynomial time by using maximum flow techniques. It is clear that such a cut is also an odd cycle cover for G, since in particular it separates c_1 and c_2 for each $c \in C$. It is not clear, though, that if there is a smaller odd cycle cover for G, then we will find it as such a cut. This is provided by the following lemma, which while somewhat technical, does not require advanced proof techniques.

Lemma 1 ([20]). *Consider a graph G with an odd cycle cover C with $|C| = k$ containing no redundant vertices, and a smaller odd cycle cover C' with $C' \cap C = \emptyset$ and $|C'| < k$. Let V_1' and V_2' be the two sides of the bipartite graph $G \setminus C'$. Then C' is a vertex cut in $H(G, C)$ between $\{c_1 \mid c \in C \cap V_1'\} \cup \{c_2 \mid c \in C \cap V_2'\}$ and $\{c_2 \mid c \in C \cap V_1'\} \cup \{c_1 \mid c \in C \cap V_2'\}$.*

That is, provided $C' \cap C = \emptyset$, we can in fact find C' as a vertex cut between the two classes of a valid partition, namely the valid partition (V_1, V_2) that can be constructed as follows: for $c \in C$, if c is on the first side of $G \setminus C'$, put c_1 into V_1 and c_2 into V_2; otherwise, put c_2 into V_1 and c_1 into V_2. For the proof we refer to Reed et al. [20].

To meet the requirement of $C' \cap C = \emptyset$, we simply enumerate all 2^k subsets $Y \subseteq C$; the sets Y are odd cycle covers for $G \setminus (C \setminus Y)$. We arrive at the following implementation of Compress-OCC.

```
Compress-OCC(G, C)
1  for each Y ⊆ C
2     do H ← Aux-Graph(G \ (C \ Y), Y)
3        for each  valid partition (Y1, Y2) of Y
4           do if ∃ vertex cut D in H between Y1 and Y2 with |D| < |Y|
5                 then return (C \ Y) ∪ D
6  return C
```

We examine every subset Y of the known odd cycle cover C. For each Y, we look for smaller odd cycle covers for G that can be constructed by replacing the vertices of Y in C by fewer new vertices from $V \setminus C$ (clearly, for any smaller odd cycle cover, such a Y must exist). Since we thereby decided to retain the vertices in $C \setminus Y$ in our odd cycle cover, we examine the graph $G' := G \setminus (C \setminus Y)$. If we now find an odd cycle cover D for G' with $|D| < |Y|$, we are done, since then $(C \setminus Y) \cup D$ is an odd cycle cover smaller than C for G. To find an odd cycle cover for G', we use its auxiliary graph H and Lemma 1.

Example. Let us now examine an example for Compress-OCC (see Fig. 1). Given is a graph G and an odd cycle cover $C = \{a, c, g\}$, marked with circles (Fig. 1 (a)). Observe that partitioning the remaining vertices into $\{b, d\}$ and $\{e, f, h\}$ induces a two-coloring in $G \setminus C$; only the bold edges conflict with this two-coloring in G. The function Compress-OCC now tries all subsets Y of C; we give two examples, first $Y = C$. We construct the auxiliary graph H (Fig. 1 (b)). Note how by selecting a suitable copy of the duplicated vertices

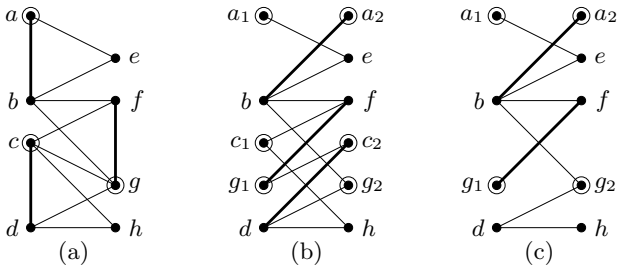

Fig. 1. Construction of auxiliary graphs in Compress-OCC. (a) G with $C = \{a, c, g\}$; (b) H for $Y = \{a, c, g\}$; (c) H for $Y = \{a, g\}$

from Y for the bold edges, we can honor the two-coloring (for example, we chose a_2 over a_1 for the edge $\{a, b\}$). The algorithm will now try to find a vertex cut of size less than 3 for some valid partition. Consider for example the valid partition $\{a_1, c_2, g_1\}$ and $\{a_2, c_1, g_2\}$. With (a_1, e, b, a_2), (c_2, d, g_2), and (g_1, f, c_1), we can find 3 vertex-disjoint paths between the two classes, so there is no vertex cut smaller than 3. In fact, for this choice of Y, there is no valid partition with a vertex cut smaller than 3. Next, we examine the case $Y = \{a, g\}$ (Fig. 1 (c)). Here we succeed: for the valid partition $\{a_1, g_1\}$, $\{a_2, g_2\}$, the set $D := \{b\}$ is a vertex cut of size 1. Note this valid partition corresponds to a two-coloring of $G \setminus ((C \setminus Y) \cup D)$. We can now construct a smaller odd cycle cover for G as $(C \setminus Y) \cup D = \{b, c\}$.

Note that although Lemma 1 does not promise it, we might also find a vertex cut that leads to a smaller odd cycle cover for some Y with $Y \cap C' \neq \emptyset$. For example, had we chosen to insert the edge $\{c_1, g_2\}$ instead of $\{c_2, g_1\}$ in Fig. 1 (b), we would have found the cut $\{b, c_1\}$ between $\{a_1, c_1, g_1\}$ and $\{a_2, c_2, g_2\}$, leading to the odd cycle cover $\{b, c\}$. Therefore, in practice one can find a smaller odd cycle cover often much faster than predicted by the worst case estimation.

Running Time. Reed et al. [20] state the run time of their algorithm as $O(4^k \cdot kmn)$; a slightly more careful analysis reveals it as $O(3^k \cdot kmn)$. For this, note that in effect the two loops in line 1 and 3 of Compress-OCC iterate over all possible assignments of each $c \in C$ to 3 roles: either $c \in C \setminus Y$, or $c \in Y_1$, or $c \in Y_2$. Therefore, we solve 3^k flow problems, and since we can solve one flow problem in $O(km)$ time by the Edmonds-Karp algorithm [2], the run time for one invocation of Compress-OCC is $O(3^k \cdot km)$. As Odd-Cycle-Cover calls Compress-OCC n times, we arrive at an overall run time of $O(3^k \cdot kmn)$.

Theorem 1. Graph Bipartization *can be solved in* $O(3^k \cdot kmn)$ *time.*

4 Algorithmic Improvements

In this section we present several improvements over the algorithm as described by Reed et al. [20]. We start with two simple improvements that save a constant

factor in the run time. In Sect. 4.1 we then show how to save a factor of k in the run time, and in Sect. 4.2 we present the improvement which gave the most pronounced speedups in our experiments presented in Sect. 5.

First, it is easy to see that each valid partition (Y_1, Y_2) is symmetric to (Y_2, Y_1) when looking for a vertex cut, and therefore we can arbitrarily fix the allocation of one vertex to Y_1, saving a factor of 2 in the run time.

The next improvement is justified by the following lemma.

Lemma 2. *Given a graph $G = (V, E)$, a vertex $v \in V$, and a minimum odd cycle cover C for $G \setminus \{v\}$ with $|C| = k$. Then no odd cycle cover of size k for G contains v.*

Proof. If C' is an odd cycle cover of size k for G, then $C' \setminus \{v\}$ is an odd cycle cover of size $k-1$ for $G[V \setminus \{v\}]$, contradicting that $|C|$ is of minimum size. □

With Lemma 2 it is clear that the vertex v we add to C in line 5 of Odd-Cycle-Cover cannot be part of a smaller odd cycle cover, and we can omit the case $v \notin Y$ in Compress-OCC, saving a third of the cases.

4.1 Exploiting Similarity of Flow Subproblems

The idea here is that the flow problems solved in Compress-OCC are "similar" in such a way that we can "recycle" the flow networks for each problem. Recall that each flow problem corresponds to one assignment of the vertices in C to the three roles "c_1 source, c_2 target" ($c \in Y_1$), "c_2 source, c_1 target" ($c \in Y_2$), and "not present" ($c \in C \setminus Y$). Using a so-called $(3, k)$-ary Gray code [10], we can enumerate these assignments in such a way that adjacent assignments differ in only one element. For each of these (but the first one), one can solve the flow problem by adapting the previous flow:

- If the affected vertex c was present previously, zero the flow along the paths with end points c_1 resp. c_2 (note they might be identical).
- If c is present in the updated assignment, find an augmenting path from c_1 to c_2 resp. from c_2 to c_1.

Since each of these operations can be done in $O(m)$ time, we can perform the update in $O(m)$ time, as opposed to $O(km)$ time for solving a flow problem from scratch. This improves the overall worst case run time to $O(3^k \cdot mn)$. We call this algorithm OCC-Gray.

Theorem 2. Graph Bipartization *can be solved in $O(3^k \cdot mn)$ time.*

4.2 Enumeration of Valid Partitions

Lemma 1 tells us that given the correct subset Y of an odd cycle cover C, there is a valid partition for Y such that we will find a cut in the auxiliary graph leading to a smaller odd cycle cover C'. Therefore, simply trying all valid partitions will

be successful. However, Lemma 1 even describes the valid partition that will lead to success: it corresponds to a two-coloring of the vertices in $G \setminus C'$. This allows us to omit some valid partitions from consideration. If for example there is an edge between two vertices $c, d \in Y$, then any two-coloring of $G \setminus C'$ must place c and d on different sides. Therefore, we only need to consider valid partitions that place c and d into different classes. This leads to the following modification of Compress-OCC:

```
Compress-OCC'(G = (V, E), C)
1  for each  bipartite subgraph B of G[C]
2     do for each  two-coloring V_1, V_2 of B
3           do H ← Aux-Graph(G \ (C \ V(B)), V(B))
4              if ∃ vertex cut D in H between V_1 and V_2 with |D| < |V(B)|
5                 then return (C \ V(B)) ∪ D
6  return C
```

The correctness of this algorithm follows directly from Lemma 1. The worst case for Compress-OCC' is that C is an independent set in G. In this case, every subgraph of $G[C]$ is bipartite and has $2^{|C|}$ two-colorings. This leads to exactly the same number of flow problems solved as for Compress-OCC. In the best case, C is a clique, and $G[C]$ has only $O(|C|^2)$ bipartite subgraphs, each of which admits (up to symmetry) only one two-coloring.

It is easy to construct a graph where any optimal odd cycle cover is independent; therefore the described modification does not lead to an improvement of the worst-case run time. However, at least in a dense graph, it is "unlikely" that the odd cycle covers are completely independent, and already a few edges between vertices of the odd cycle cover can vastly reduce the required computation.

Note that with a simple branching strategy, one can enumerate all bipartite subgraphs of a graph and all their two-colorings with constant cost per two-coloring. This can also be done in such a way that modifications to the flow graph can be done incrementally, as described in Sect. 4.1. The two simple improvements mentioned at the beginning of this section also can still be applied. We call the thus modified algorithm OCC-Enum2Col.

It seems plausible that for dense graphs, an odd cycle cover is "more likely" to be connected, and therefore this heuristic is more profitable. Experiments on random graphs confirm this (see Sect. 5.3). This is of particular interest because other strategies (such as reduction rules [22]) seem to have a harder time with dense graphs than with sparse graphs, making hybrid algorithms appealing.

5 Experiments

Implementation Details. The program is written in the C programming language and consists of about 1400 lines of code. The source and the test data are available from http://www.minet.uni-jena.de/~hueffner/occ.

Data structures. Over 90% of the time is spent in finding an augmenting path within the flow network; all that this requires from a graph data structure is enumerating the neighbors of a given vertex. The only other frequent operation is "enabling" or "disabling" vertices as determined by the Gray code (see Sect. 4.1). In particular, it is not necessary to quickly add or remove edges, or query whether two vertices are neighbored. Therefore, we chose a very simple data structure, where the graph is represented by an array of neighbor lists, with a null pointer denoting a disabled vertex.

Since the flow simply models a set of vertex-disjoint paths, it is not necessary to store a complete $n \times n$-matrix of flows; it suffices to store the flow predecessor and successor for each node, reducing memory usage to $O(n)$.

Finding Vertex Cuts. It has now become clear that in the "inner loop" of the algorithm, we need to find a minimum vertex cut between two sets Y_1 and Y_2 in a graph G, or equivalently, a maximum set of vertex-disjoint paths between two sets. This is a classical application for maximum flow techniques: The well-known max-flow min-cut theorem tells us that the size of a minimum edge cut is equal to the maximum flow. Since we are interested in vertex cuts, we create a new, directed graph G' for our input graph $G = (V, E)$: for each vertex $v \in V$, create two vertices v_{in} and v_{out} and a directed edge (v_{in}, v_{out}). For each edge $\{v, w\} \in E$, we add two directed edges (v_{out}, w_{in}) and (w_{out}, v_{in}). It is not too hard to see that a maximum flow in G' between $Y_1' := \bigcup_{y \in Y_1} y_{in}$ and $Y_2' := \bigcup_{y \in Y_2} y_{out}$ corresponds to a maximum set of vertex disjoint paths between Y_1 and Y_2. Furthermore, an edge cut D between Y_1' and Y_2' is of the form $\bigcup_{v \in V} (v_{in}, v_{out})$, and $\bigcup_{(v_{in}, v_{out}) \in D} v$ is a vertex cut between Y_1 and Y_2 in G.

Since we know that the cut is relatively small (less than or equal k), we employ the Edmonds-Karp algorithm [2]. This algorithm repeatedly finds a shortest augmenting path in the flow network and increases the flow along it, until no further increase is possible.

Experimental Setup. We tested our implementation on various inputs. The testing machine is an AMD Athlon 64 3400+ with 2400 MHz, 512 KB cache, and 1 GB main memory, running under the Debian GNU/Linux 3.1 operating system. The source was compiled with the GNU gcc 3.4.3 compiler with options "-O3 -march=k8". Memory requirements are around 3 MB for the iterative compression based algorithms, and up to 500 MB for the ILP.

5.1 Minimum Site Removal

The first test set originates from computational biology. The instances were constructed by Wernicke [22] from data of the human genome as a means to solve the so-called Minimum Site Removal problem. The results are shown in Table 1.

As expected, the run time of the iterative compression algorithms mainly depends on the size of the odd cycle cover that is to be found. Interestingly, the ILP also shows this behavior. The observed improvement in the run time

Table 1. Run times in seconds for different algorithms for Wernicke's benchmark instances [22]. Runs were cancelled after 2 hours without result. We show only the instance of median size for each value of $|C|$. The column "ILP" gives the run time of the ILP given in Sect. 2 when solved by GNU GLPK [15]. The column "Reed" gives the run time of Reed et al.'s algorithm without any of the algorithmic improvements from Sect. 4 except for omitting symmetric valid partitions. The columns "OCC-Gray" and "OCC-Enum2Col" give the run time for the respective algorithms from Sect. 4.1 and 4.2. The "augmentations" colums give the number of flow augmentations performed.

	n	m	$\|C\|$	ILP	Reed		OCC-Gray		OCC-Enum2Col	
				time [s]	time [s]	augmentations	time [s]	augmentations	time [s]	augmentations
Afr. #31	30	51	2	0.02	0.00	7	0.00	6	0.00	5
Jap. #19	84	172	3	0.12	0.00	27	0.00	14	0.00	10
Jap. #24	142	387	4	0.97	0.00	117	0.00	46	0.00	31
Jap. #11	51	212	5	0.46	0.00	412	0.00	109	0.00	79
Afr. #10	69	191	6	2.50	0.00	1,558	0.00	380	0.00	97
Afr. #36	111	316	7	15.97	0.01	5,109	0.00	696	0.00	1,392
Jap. #18	71	296	9	47.86	0.05	59,052	0.01	7,105	0.00	568
Jap. #17	79	322	10	237.16	0.22	205,713	0.02	18,407	0.00	1,591
Afr. #11	102	307	11	6248.12	0.79	671,088	0.14	85,851	0.00	1,945
Afr. #54	89	233	12		6.48	5,739,277	0.73	628,445	0.03	20,385
Afr. #34	133	451	13		10.13	6,909,386	1.04	554,928	0.04	16,413
Afr. #52	65	231	14		18.98	22,389,052	1.83	2,037,727	0.01	11,195
Afr. #22	167	641	16		350.00	229,584,280	64.88	15,809,779	0.08	22,607
Afr. #48	89	343	17		737.24	731,807,698	74.20	54,162,116	0.06	41,498
Afr. #50	113	468	18		3072.82	2,913,252,849	270.60	151,516,435	0.05	26,711
Afr. #19	191	645	19				1020.22	421,190,990	3.70	1,803,293
Afr. #45	80	386	20				2716.87	2,169,669,374	0.14	99,765
Afr. #29	276	1058	21						0.23	56,095
Afr. #40	136	620	22						0.80	333,793
Afr. #39	144	692	23						0.65	281,403
Afr. #17	151	633	25						5.68	2,342,879
Afr. #38	171	862	26						1.69	631,053
Afr. #28	167	854	27						1.02	464,272
Afr. #42	236	1110	30						73.55	22,588,100
Afr. #41	296	1620	40						236.26	55,758,998

from "Reed" to "OCC-Gray" is lower than the factor of k gained in the worst case complexity, but clearly still worthwhile. The heuristic from Sect. 4.2 works exceedingly well and allows to solve even the hardest instances within minutes. For both improvements, the savings in run time closely follow the savings of flow augmentations.

5.2 Synthetic Data from Computational Biology

In this section we examine solving the Minimum Fragment Removal [18] problem with Graph Bipartization. We generate synthetic Graph Bipartization instances using a model of Panconesi and Sozi [18], with parameters $n = 100$, $d = 0.2$, $k = 20$, $p = 0.02$, and c varying (see Table 2). We refer to [18] for details on the model and its parameters.

The results are consistent with those of Sect. 5.1. The ILP is outperformed by the iterative compression algorithms; for OCC-Gray, we get a speedup by a factor somewhat below $|C|$ when compared to "Reed". The speedup from employing OCC-Enum2Col is very pronounced, but still far below the speedup observed in Sect. 5.1. A plausible explanation is the lower average vertex degree of the input instances; we examine this further in Sect. 5.3. Note that even with all model parameters constant, run times varied by a factor of up to several orders of magnitude for all algorithms for different random instances.

Table 2. Run times in seconds for different algorithms for synthetic Minimum Fragment Removal instances [18]. Here, c is a model parameter. Average over 20 instances each.

c	$\|V\|$	$\|E\|$	$\|C\|$	ILP	Reed	OCC-Gray	OCC-Enum2Col
2	24	22	1.4	0.02	0.00	0.00	0.00
3	49	58	3.1	1.40	0.00	0.00	0.00
4	75	103	4.8	1538.41	0.02	0.00	0.00
5	111	169	7.7		4.18	0.42	0.04
6	146	247	9.8		5.22	0.68	0.04
7	181	353	13.8		3044.25	238.80	1.89
8	214	447	14.9			4547.54	8.03
9	246	548	16.8				17.41
10	290	697	20.1				744.19

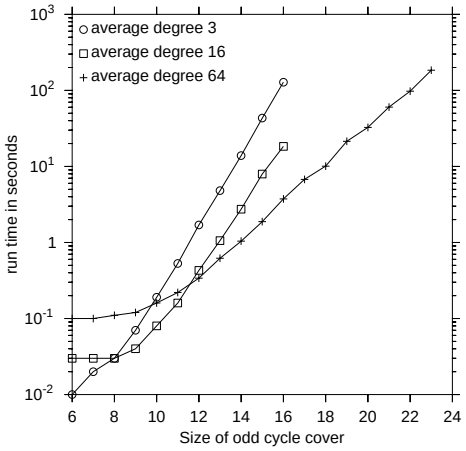

Fig. 2. Run time of OCC-Enum2Col (Sect. 4.2) for random graphs of different density ($n = 300$). Each point is the average over at least 40 runs.

5.3 Random Graphs

The previous experiments have established OCC-Enum2Col as a clear winner. Therefore, we now focus on charting its tractability border. We use the following method to generate random graphs with given number of vertices n, edges m, and odd cycle cover size at most k: Pre-allocate the roles "black" and "white" to $(n-k)/2$ vertices each, and "odd cycle cover" to k vertices; select a random vertex and add an edge to another random vertex consistent with the roles until m edges have been added.

In Fig. 2, we display the run time of OCC-Enum2Col for different sizes of the odd cycle cover and different graph densities for graphs with 300 vertices. Note that the actual optimal odd cycle cover can be smaller than the one "implanted" by our model; the figure refers to the actual odd cycle cover size k.

At an average degree of 3, the growth in the measurements closely matches the one predicted by the worst-case complexity $O(3^k)$. For the average degree 16, the measurements fit a growth of $O(2.5^k)$, and for average degree 64, the growth within the observed range is about $O(1.7^k)$. This clearly demonstrates the effec-

tiveness of OCC-Enum2Col for dense graphs, at least in the range of values of k we examined.

6 Conclusions

We evaluated the iterative compression algorithm by Reed et al. [20] for Graph Bipartization and presented several improvements. The implementation performs better than established techniques, and allows to solve instances from computational biology that previously could not be solved exactly. In particular, a heuristic (Sect. 4.2) yielding optimal solutions performs very well on dense graphs. This result makes the practical evaluation of iterative compression for other applications [3, 4, 11, 16] appealing.

Future Work.

- Wernicke [22] reports that data reduction rules are most effective for sparse graphs. This makes a combination with OCC-Enum2Col (Sect. 4.2) attractive, since in contrast, this algorithm displays the worst performance for sparse graphs.
- Guo et al. [11] give an $O(2^k \cdot km^2)$ time algorithm for Edge Bipartization, where the task is to remove up to k edges from a graph to make it bipartite. The algorithm is based on iterative compression; it would be interesting to see whether our improvements can be applied here, and do experiments with real world data.
- Iterative compression can also be employed to "compress" a non-optimal solution until an optimal one is found. Initial experiments indicate that OCC-Enum2Col with this mode finds an optimal solution very quickly, even when starting with $C = V$, but then takes a long time to prove the optimality.

Acknowledgements. The author is grateful to Jens Gramm (Tübingen) and Jiong Guo, Rolf Niedermeier, and Sebastian Wernicke (Jena) for many helpful suggestions and improvements.

References

1. H.-A. Choi, K. Nakajima, and C. S. Rim. Graph bipartization and via minimization. *SIAM Journal on Discrete Mathematics*, 2(1):38–47, 1989.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.
3. F. Dehne, M. R. Fellows, M. A. Langston, F. A. Rosamond, and K. Stevens. An $\mathcal{O}^*(2^{O(k)})$ FPT algorithm for the undirected feedback vertex set problem. Manuscript, Dec. 2004.
4. F. Dehne, M. R. Fellows, F. A. Rosamond, and P. Shaw. Greedy localization, iterative compression, and modeled crown reductions: New FPT techniques, an improved algorithm for set splitting, and a novel $2k$ kernelization for Vertex Cover. In *Proc. 1st IWPEC*, volume 3162 of *LNCS*, pages 271–280. Springer, 2004.

5. R. G. Downey. Parameterized complexity for the skeptic. In *Proc. 18th IEEE Annual Conference on Computational Complexity*, pages 147–169, 2003.
6. R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.
7. M. R. Fellows. Blow-ups, win/win's, and crown rules: Some new directions in FPT. In *Proc. 29th WG*, volume 2880 of *LNCS*, pages 1–12. Springer, 2003.
8. M. R. Fellows. New directions and new challenges in algorithm design and complexity, parameterized. In *Proc. 8th WADS*, volume 2748 of *LNCS*, pages 505–520. Springer, 2003.
9. N. Garg, V. V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi)cut theorems and their applications. *SIAM Journal on Computing*, 25(2):235–251, 1996.
10. D.-J. Guan. Generalized Gray codes with applications. *Proceedings of the National Science Council, Republic of China (A)*, 22(6):841–848, 1998.
11. J. Guo, J. Gramm, F. Hüffner, R. Niedermeier, and S. Wernicke. Improved fixed-parameter algorithms for two feedback set problems. Manuscript, Feb. 2005.
12. A. B. Kahng, S. Vaya, and A. Zelikovsky. New graph bipartizations for double-exposure, bright field alternating phase-shift mask layout. In *Proc. Asia and South Pacific Design Automation Conf.*, pages 133–138. ACM, 2001.
13. J. M. Lewis and M. Yannakakis. The node-deletion problem for hereditary properties is NP-complete. *Journal of Computer and System Sciences*, 20(2):219–230, 1980.
14. M. Mahajan and V. Raman. Parameterizing above guaranteed values: MaxSat and MaxCut. *Journal of Algorithms*, 31(2):335–354, 1999.
15. A. Makhorin. *GNU Linear Programming Kit Reference Manual Version 4.7*. Dept. Applied Informatics, Moscow Aviation Institute, 2004.
16. D. Marx. Chordal deletion is fixed-parameter tractable. Manuscript, Dept. Computer Science, Budapest University of Technology and Economics, Aug. 2004.
17. R. Niedermeier. Ubiquitous parameterization—invitation to fixed-parameter algorithms. In *Proc. 29th MFCS*, volume 3153 of *LNCS*, pages 84–103. Springer, 2004.
18. A. Panconesi and M. Sozio. Fast hare: A fast heuristic for single individual SNP haplotype reconstruction. In *Proc. 4th WABI*, volume 3240 of *LNCS*, pages 266–277. Springer, 2004.
19. C. H. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.
20. B. Reed, K. Smith, and A. Vetta. Finding odd cycle transversals. *Operations Research Letters*, 32(4):299–301, 2004.
21. R. Rizzi, V. Bafna, S. Istrail, and G. Lancia. Practical algorithms and fixed-parameter tractability for the single individual SNP haplotyping problem. In *Proc. 2nd WABI*, LNCS, pages 29–43. Springer, 2002.
22. S. Wernicke. On the algorithmic tractability of single nucleotide polymorphism (SNP) analysis and related problems. Diplomarbeit, Univ. Tübingen, Sept. 2003.
23. G. J. Woeginger. Exact algorithms for NP-hard problems: A survey. In *Proc. 5th International Workshop on Combinatorial Optimization*, volume 2570 of *LNCS*, pages 185–208. Springer, 2003.
24. X. Zhuang and S. Pande. Resolving register bank conflicts for a network processor. In *Proc. 12th PACT*, pages 269–278. IEEE Press, 2003.