WeGotYouCovered*

Demian Hespe† Sebastian Lamm‡ Christian Schulz§ Darren Strash¶

Abstract

We present the winning solver of the PACE 2019 Implementation Challenge Vertex Cover Track. The vertex cover problem is one of a handful of problems for which *kernelization*—the repeated reducing of the input size via *data reduction rules*—is known to be highly effective in practice. Our algorithm uses a portfolio of techniques, including an aggressive kernelization strategy with all known reduction rules, local search, branch-and-reduce, and a state-of-the-art branch-and-bound solver. Of particular interest is that several of our techniques were *not* from the literature on the vertex over problem: they were originally published to solve the (complementary) maximum independent set and maximum clique problems. Lastly, we perform extensive experiments to show the impact of the different solver techniques on the number of instances solved during the challenge.

1 Introduction

A *vertex cover* of a graph $G = (V, E)$ is a set of vertices $S \subseteq V$ of G such that every edge of G has at least one of member of S as an endpoint (i.e., $\forall (u, v) \in E\ [u \in S \text{ or } v \in S]$). A minimum vertex cover is a vertex cover of minimum cardinality. Complementary to vertex covers are independent sets and cliques. An independent set is a set of vertices $I \subseteq V$, all pairs of which are not adjacent, and an clique is a set of vertices $K \subseteq V$ all pairs of which are adjacent. A maximum independent set (maximum clique) is an independent set (clique) of maximum cardinality. The goal of the maximum independent set problem (maximum clique problem) is to compute a maximum independent set (maximum clique).

Many techniques have been proposed for solving these problems, and papers in the literature usually focus on one of these problems in particular. However, all of these problems are equivalent: a minimum vertex cover C in G is the complement of a maximum independent set $V \setminus C$ in G, which is a maximum clique $V \setminus C$ in $\overline{G}$. Thus, an algorithm that solves one of these problems can be used to solve the others. For our approach, we use a portfolio of solvers, using techniques from the literature on all three problems. These include data reduction rules and branch-and-reduce for the minimum vertex cover problem [2], iterated local search for the maximum independent set problem [3], and a state-of-the-art branch-and-bound maximum clique solver [14].

We first briefly describe releated work. Then we outline each of the techniques that we use, and finally describe how we combine all of the techniques in our final solver that scored most of the points during the PACE 2019 Implementation Challenge. Lastly, we perform an experimental evaluation to show the impact of the components used on the final number of instances solved during the challenge.

2 Related Work

We now present important related work. This includes exact branch-and-bound algorithms as well as reduction based approaches. Much research has been devoted to improve exact branch-and-bound algorithms for the independent set and its complementary problems. These improvements include different pruning methods and sophisticated branching schemes [16, 5, 4, 18]. Warren and Hicks [18] proposed three combinatorial branch-and-bound algorithms that are able to quickly solve DIMACS and weighted random graphs. These algorithms use weighted clique covers to generate upper bounds that reduce the search space via pruning. Furthermore, they all use a branching scheme proposed by Balas and Yu [5]. In particular, their first algorithm is an extension and improvement of a method by Babel [4]. Their second one uses a modified version of the algorithm by Balas and Yu that uses clique covers that borrow structural features from the ones by Babel [4]. Finally, their third approach is a hybrid of both previous algorithms. Overall, their algorithms are able to quickly solve instances with hundreds of vertices.

An important technique to reduce the base of the exponent for exact branch-and-bound algorithms are

*The research leading to these results has received funding from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) /ERC grant agreement No. 340506

†Karlsruhe Institute of Technology, Karlsruhe, Germany

‡Karlsruhe Institute of Technology, Karlsruhe, Germany

§University of Vienna, Faculty of Computer Science, Austria

¶Hamilton College, New York, USA

so-called *reduction rules*. Reduction rules are able to reduce the input graph to an irreducible *kernel* by removing well-defined subgraphs. This is done by selecting certain vertices that are provably part of some maximum independent set, thus maintaining optimality. We can then extend a solution on the kernel to a solution on the input graph by undoing the previously applied reductions. There exist several well-known reduction rules for the unweighted vertex cover problem (and in turn for the unweighted MIS problem) [2].

As noted by Larson [13], it is possible that in the unweighted case the initial critical set found by Butenko and Trukhanov might be empty. To prevent this case, Larson [13] proposed an algorithm that finds a *maximum* (unweighted) critical independent set. His algorithm accumulates vertices that are in some critical set and removes their neighborhood. Additionally, he provides a method to quickly check if a given vertex is part of some critical set. Later Iwata [12] has shown how to remove a large collection of vertices from a maximum matching all at once; however, it is not known if these reductions are equivalent.

For the maximum weight clique problem, Cai and Lin [8] give an exact branch-and-bound algorithm that interleaves between clique construction and reductions. In particular, their algorithm picks different starting vertices to form a clique and then maintains a candidate set to iteratively extend this clique. In each iteration, the vertex to be added is selected using a benefit estimation function and a dynamic best from multiple selection heuristic [7]. Once the candidate set is empty, the new solution is compared to the best solution found so far. If an improvement is found, their algorithm then tries to apply reductions and reduce the graph size. To be more specific, they use two reduction rules that are able to remove a vertex v by computing upper bounds related to the weight of different neighborhoods of v. We briefly note that their algorithm and reductions are targeted at sparse graphs, and therefore their reductions would likely work well for the maximum weighted independent set problem on *dense* graphs.

3 Techniques

We now describe techniques that we use in our solver.

3.1 Kernelization. The most efficient algorithms for computing a minimum vertex cover in both theory and practice use *data reduction rules* to obtain a much smaller problem instance. If this smaller instance has size bounded by a function of some parameter, it's called a *kernel*.

We use an extensive (though not exhaustive) collection of data reduction rules whose efficacy was studied by Akiba and Iwata [2]. To compute a kernel, Akiba and Iwata [2] apply their reductions $r_1, \ldots, r_j$ by iterating over all reductions and trying to apply the current reduction r_i to all vertices. If r_i reduces at least one vertex, they restart with reduction r_1. When reduction r_j is executed, but does not reduce any vertex, all reductions have been applied exhaustively, and a kernel is found. Following their study we order the reductions as follows: degree-one vertex (i.e., pendant) removal, vertex folding [10], a well-known linear-programming relaxation [12, 15] related to crown removal [1], unconfined vertex removal [19], and twin, funnel, and desk reductions [19]. To be self-contained, we now give a brief description of those reductions, in order of increasing complexity. Each reduction allows us to choose vertices that are in some MIS by following simple rules. If an MIS is found on the kernel graph $\mathcal{K}$, then each reduction may be undone, producing an MIS in the original graph. Refer to Akiba and Iwata [2] for a more thorough discussion, including implementation details. We use our own implementation of the reduction algorithms in our method.

Pendant vertices: Any vertex v of degree one, called a *pendant*, is in some MIS; therefore v and its neighbor u can be removed from G.

Vertex folding: For a vertex v with degree 2 whose neighbors u and w are not adjacent, either v is in some MIS, or both u and w are in some MIS. Therefore, we can contract u, v, and w to a single vertex v' and decide which vertices are in the MIS later.

Linear Programming: First introduced by Nemhauser and Trotter [15] for the vertex packing problem, they present a linear programming relaxation with a half-integral solution (i.e., using only values 0, 1/2, and 1) which can be solved using bipartite matching. Their relaxation may be formulated for the independent set problem as follows: maximize $\sum_{v \in V} x_v$, such at for each edge $(u, v) \in E$, $x_u + x_v \leq 1$ and for each vertex $v \in V$, $x_v \geq 0$. Vertices with value 1 must be in the MIS, and therefore are added to the solution. We use the further improvement from Iwata, Oka, and Yoshida [12], which computes a solution whose half-integral part is minimal.

Unconfined: Developed by Xiao and Nagamochi [19], the unconfined reduction is a generalization of domination and *satellite* reduction rules. A vertex v is said to be unconfined if there exists a set S, such that $v \in S$ and $\exists u \in S$ such that $|N(u) \cap S| = 1$ and $N(u) \setminus N[S]$ is empty. Such a vertex is never in a MIS, so it can be removed from the graph.

Twin: This is a generalization of the vertex folding rule. Suppose there are two vertices u and v that have degree 3 and share the same neighborhood. If u's neighborhood $N(u)$ induces a graph with edges, then u and v are added to the independent set and u, v, and their neighborhoods are removed from the graph. Otherwise, vertices in $N(u)$ may belong in the independent set. We still remove u, v, and their neighborhoods, and add a new gadget vertex w to the graph with edges to u's two-neighborhood (vertices at a distance 2 from u). If w is in some MIS, none of u's two-neighbors are in the independent set, and therefore $N(u)$ is part of the independent set. Otherwise, if w is not in the MIS, then some of u's two-neighbors are in the independent set, and therefore u and v are added to the independent set. Thus, the twin reduction adds an additional two vertices to the computed independent set.

Alternative: Two sets of vertices A and B are set to be *alternatives* if $|A| = |B| \geq 1$ and there exists an MIS $\mathcal{I}$ such that $\mathcal{I} \cap (A \cup B)$ is either A or B. Then we remove A and B and $C = N(A) \cap N(B)$ from G and add edges from each $a \in N(A) \setminus C$ to each $b \in N(B) \setminus C$. Then we add either A or B to $\mathcal{I}$, depending on which neighborhood has vertices in $\mathcal{I}$. Two structures are detected as alternatives. First, if $N(v) \setminus \{u\}$ induces a complete graph, then $\{u\}$ and $\{v\}$ are alternatives (a *funnel*). Next, if there is a cordless 4-cycle $a_1b_1a_2b_2$ where each vertex has at least degree 3. Then sets $A = \{a_1, a_2\}$ and $B = \{b_1, b_2\}$ are alternatives when $|N(A) \setminus B| \leq 2$, $|N(A) \setminus B| \leq 2$, and $N(A) \cap N(B) = \emptyset$.

Packing [2]: Given a non-empty set of vertices S, we may specify a *packing constraint* $\sum_{v \in S} x_v \leq k$, where x_v is 0 when v is in some MIS $\mathcal{I}$ and 1 otherwise. Whenever a vertex v is excluded from $\mathcal{I}$ (i.e., in the unconfined reduction), we remove x_v from the packing constraint and decrease the upper bound of the constraint by one. Initially, packing constraints are created whenever a vertex v is excluded or included into the MIS. The simplest case for the packing reduction is when k is zero: all vertices must be in $\mathcal{I}$ to satisfy the constraint. Thus, if there is no edge in $G[S]$, S may be added to $\mathcal{I}$, and S and $N(S)$ are removed from G. Other cases are much more complex. Whenever packing reductions are applied, existing packing constraints are updated and new ones are added.

3.2 Branch-and-Reduce. Branch-and-reduce is a paradigm that intermixes data reduction rules and branching. We use the algorithm of Akiba and Iwata, which exhaustively applies their full suite of reduction rules before branching, and includes a number of advanced branching rules. When branching, a vertex is chosen at random for inclusion into the vertex cover.

3.3 Branch-and-Bound. Experiments by Strash [17] show that the full power of branch-and-reduce is only needed *very rarely* in real-world instances; kernelization followed by standard branch-and-bound solver is sufficient for many real-world instances. Furthermore, branch-and-reduce does not work well on many synthetic benchmark instances, where data reduction rules are ineffective [2], and instead add significant overhead to branch-and-bound. We use a state-of-the-art branch-and-bound maximum clique solver (MoMC) by Li et al. [14], which uses incremental MaxSAT reasoning to prune search, and a combination of static and dynamic vertex ordering to select the vertex for branching. We run the clique solver on the complement graph, giving a maximum independent set from which we derive a minimum vertex cover. In preliminary experiments, we found that a kernel can sometimes be harder for the solver than the original input; therefore, we run the algorithm on both the kernel and on the original graph.

3.4 Iterated Local Search. Batsyn et al. [6] showed that if branch-and-bound search is primed with a high-quality solution from local search, then instances can be solved up to thousands of times faster. We use iterated local search algorithm by Andrade et al. [3] to prime the branch-and-reduce solver with a high-quality initial solution. Iterated local search was originally implemented for the maximum independent set problem, and is based on the notion of (j, k)-swaps. A (j, k)-swap removes j nodes from the current solution and inserts k nodes. The authors present a fast linear-time implementation that, given a maximal independent set, can find a $(1, 2)$-swap or prove that none exists. Their algorithm applies $(1, 2)$-swaps until reaching a local maximum, then perturbs the solution and repeats. We implemented the algorithm to find a high-quality solution on *the kernel*. Calling local search on the kernel has been shown to produce a high-quality solution much faster than without kernelization [9, 11].

4 Putting it all Together

Our algorithm first runs a preprocessing phase, followed by 4 phases of solvers.

Phase 1. (Preprocessing) Our algorithm starts by computing a kernel of the graph using the reductions by Akiba and Iwata [2]. From there we use iterated local search to produce a high-quality solution S_{init} on the (hopefully smaller) kernel.

Phase 2. (Branch-and-Reduce, short) We prime a branch-and-reduce solver with the initial solution S_{init} and run it with a short time limit.

Phase 3. (Branch-and-Bound, short) If Phase 2 is unsuccessful, we run the MoMC [14] clique solver on the complement of the kernel, also using a short time limit. Sometimes kernelization can make the problem harder for MoMC. Therefore, if the first call was unsuccessful we also run MoMC on the complement of the original (unkernelized) input with the same short time limit.

Phase 4. (Branch-and-Reduce, long) If we have still not found a solution, we run branch-and-reduce on the kernel using initial solution S_{init} and a longer time limit. We opt for this second phase because, while most graphs amenable to reductions are solved very quickly with branch-and-reduce (less than a second), experiments by Akiba and Iwata [2] showed that other slower instances either finish in at most a few minutes, or take significantly longer—more than the time limit allotted for the challenge. This second phase of branch-and-reduce is meant to catch any instances that still benefit from reductions.

Phase 5. (Branch-and-Bound, remaining time) If all previous phases were unsuccessful, we run MoMC on the original (unkernelized) input graph until the end of the time given to the program by the challenge. This is meant to capture only the most hard-to-compute instances.

The ordering and time limits were carefully chosen so that the overall algorithm outputs solutions of the "easy" instances *quickly*, while still being able to solve hard instances.

5 Experimental Results

We now look at the impact of the algorithmic components on the number of instances solved. Here, we use the public instances – obtained from `https://pacechallenge.org/files/pace2019-vc-exact-public-v2.tar.bz2` – of the PACE 2019 Track A implementation challenge. This set contains 100 instances overall. Afterwards, we present the results comaring against the second and third best competing algorithms during the challenge.

5.1 Methodology and Setup. All of our experiments were run on a machine with four Sixteen-Core Intel Xeon Haswell-EX E7-8867 processors running at 2.5 GHz, 1 TB of main memory, and 32768 KB L2-Cache. The machine runs Debian GNU/Linux 9 and Linux kernel version 4.9.0-9. All algorithms were implemented in C++11 and compiled with gcc version 6.3.0 with optimization flag `-O3`. Each algorithm was run sequentially with a time limit of 30 minutes. Our evaluations focus on the total number of instances solved.

5.2 Evaluation. We now explain our main configuration that we use in our experimental setup. In the following `MoMC` runs the clique solver [14] on the complement of the input graph, `RMoMC` applies reductions to the input graph exhaustively and then runs MoMC on the complement of the kernel graph, `BnR` applies reductions exhaustively, then runs local search to obtain a high-quality solution on the kernel which is used as a initial bound in the branch-and-reduce algorithm that is run on the kernel, `BnR-LS` applies reductions and then runs the branch-and-reduce algorithm on the kernel (no local search is used to improve an initial bound), `FullA` is the full algorithm as described above.

Tables 1 and 2 give an overview over the instances that each of the solver solved, about the kernel sizes as well as the optimal vertex cover size, if our full algorithm could solve the instance. Overall, `MoMC` can solve 30 out of the 100 instances. Using reductions first, enables `RMoMC` to solve 68 instances. However, there are also instances that `MoMC` could solve, but `RMoMC` could not solve. In these case, the number of nodes has been reduced, but the number of edges actually increased. This is due to the *Alternative* reduction, which in some cases can create more edges than initially present. This is why our full algorithm also runs MoMC on the input graph (in order to be able to solve those instances as well). `BnR` can solve 55 out of the 100 instances. Here, priming the branch-and-reduce algorithm with an initial solution computed by local search has an important impact. Running the branch-and-reduce algorithm on the kernel without using local search can only solve 42

Table 1: Detailed per instance results. The columns n,m refer to the number of nodes and edges of the input graph, n',m' refer to the number of nodes and edges of the kernel graph after reductions have been applied exhaustively, and $|VC|$ refers to the size of the optimal vertex cover of the input graph. With X we donote if a solver has been solving an instance, and with - we denote if this has not been the case.

inst#	n	m	n'	m'	MoMC	RMoMC	BnR	BnR-LS	FullA	$\|VC\|$
001	6 160	40 207	0	0	-	X	X	X	X	2 586
003	60 541	74 220	0	0	-	X	X	X	X	12 190
005	200	819	192	800	X	X	X	X	X	129
007	8 794	10 130	0	0	-	X	X	X	X	4 397
009	38 452	174 645	0	0	-	X	X	X	X	21 348
011	9 877	25 973	0	0	-	X	X	X	X	4 981
013	45 307	55 440	0	0	-	X	X	X	X	8 610
015	53 610	65 952	0	0	-	X	X	X	X	10 670
017	23 541	51 747	0	0	-	X	X	X	X	12 082
019	200	884	194	862	X	X	X	X	X	130
021	24 765	30 242	0	0	-	X	X	X	X	5 110
023	27 717	133 665	0	0	-	X	X	X	X	16 013
025	23 194	28 221	0	0	-	X	X	X	X	4 899
027	65 866	81 245	0	0	-	X	X	X	X	13 431
029	13 431	21 999	0	0	-	X	X	X	X	6 622
031	200	813	198	818	X	X	X	X	X	136
033	4 410	6 885	138	471	-	X	X	X	X	2 725
035	200	884	189	859	X	X	X	X	X	133
037	198	824	194	810	X	X	X	X	X	131
039	6 795	10 620	219	753	-	X	X	X	X	4 200
041	200	1 040	200	1 023	X	X	X	X	X	139
043	200	841	198	844	X	X	X	X	X	139
045	200	1 044	200	1 020	X	X	X	X	X	137
047	200	1 120	198	1 080	X	X	X	X	X	140
049	200	957	198	930	X	X	X	X	X	136
051	200	1 135	200	1 098	X	X	X	X	X	140
053	200	1 062	200	1 026	X	X	X	X	X	139
055	200	958	194	938	X	X	X	X	X	134
057	200	1 200	197	1 139	X	X	X	X	X	142
059	200	988	193	954	X	X	X	X	X	137
061	200	952	198	914	X	X	X	X	X	135
063	200	1 040	200	1 011	X	X	X	X	X	138
065	200	1 037	200	1 011	X	X	X	X	X	138
067	200	1 201	200	1 174	X	X	X	X	X	143
069	200	1 120	196	1 077	X	X	X	X	X	140
071	200	984	200	952	X	X	X	X	X	136
073	200	1 107	200	1 078	X	X	X	X	X	139
075	26 300	41 500	500	3 000	-	-	X	-	X	16 300
077	200	988	193	954	X	X	X	X	X	137
079	26 300	41 500	500	3 000	-	-	X	-	X	16 300
081	199	1 124	197	1 087	X	X	X	X	X	141
083	200	1 215	198	1 182	X	X	X	X	X	144
085	11 470	17 408	3 539	25 955	-	-	-	-	-	
087	13 590	21 240	441	1 512	-	X	-	-	X	8 400
089	57 316	77 978	16 834	54 847	-	-	-	-	-	
091	200	1 196	200	1 163	X	X	X	X	X	145
093	200	1 207	200	1 162	X	X	X	X	X	143
095	15 783	24 663	510	1 746	-	X	-	-	X	9 755
097	18 096	28 281	579	1 995	-	X	-	-	X	11 185
099	26 300	41 500	500	3 000	-	-	X	-	X	16 300

Table 2: Detailed per instance results. The columns n,m refer to the number of nodes and edges of the input graph, n',m' refer to the number of nodes and edges of the kernel graph after reductions have been applied exhaustively, and $|VC|$ refers to the size of the optimal vertex cover of the input graph. With X we donote if a solver has been solving an instance, and with - we denote if this has not been the case.

inst#	n	m	n'	m'	MoMC	RMoMC	BnR	BnR-LS	FullA	$\|VC\|$
101	26 300	41 500	500	3 000	-	-	X	-	X	16 300
103	15 783	24 663	513	1 752	-	X	-	-	X	9 755
105	26 300	41 500	500	3 000	-	-	X	-	X	16 300
107	13 590	21 240	435	1 500	-	X	-	-	X	8 400
109	66 992	90 970	20 336	66 350	-	-	-	-	-	
111	450	17 831	450	17 831	X	X	-	-	X	420
113	26 300	41 500	500	3 000	-	-	X	-	X	16 300
115	18 096	28 281	573	1 986	-	X	-	-	X	11 185
117	18 096	28 281	582	2 007	-	X	-	-	X	11 185
119	18 096	28 281	588	2 016	-	X	-	-	X	11 185
121	18 096	28 281	579	1 998	-	X	-	-	X	11 185
123	26 300	41 500	500	3 000	-	-	X	-	X	16 300
125	26 300	41 500	500	3 000	-	-	X	-	X	16 300
127	18 096	28 281	582	2 001	-	X	-	-	X	11 185
129	15 783	24 663	507	1 752	-	X	-	-	X	9 755
131	2 980	5 360	2 179	6 951	X	-	-	-	X	1 920
133	15 783	24 663	507	1 746	-	X	-	-	X	9 755
135	26 300	41 500	500	3 000	-	-	X	-	X	16 300
137	26 300	41 500	500	3 000	-	-	X	-	X	16 300
139	18 096	28 281	579	1 995	-	X	-	-	X	11 185
141	18 096	28 281	576	1 995	-	X	-	-	X	11 185
143	18 096	28 281	582	2 001	-	X	-	-	X	11 185
145	18 096	28 281	576	1 989	-	X	-	-	X	11 185
147	18 096	28 281	567	1 974	-	X	-	-	X	11 185
149	26 300	41 500	500	3 000	-	-	X	-	X	16 300
151	15 783	24 663	501	1 728	-	X	-	-	X	9 755
153	29 076	45 570	2 124	16 266	-	-	-	-	-	
155	26 300	41 500	500	3 000	-	-	X	-	X	16 300
157	2 980	5 360	2 169	6 898	X	-	-	-	X	1 920
159	18 096	28 281	582	2 004	-	X	-	-	X	11 185
161	138 141	227 241	41 926	202 869	-	-	-	-	-	
163	18 096	28 281	582	2 004	-	X	-	-	X	11 185
165	18 096	28 281	576	1 995	-	X	-	-	X	11 185
167	15 783	24 663	510	1 746	-	X	-	-	X	9 755
169	4 768	8 576	3 458	11 014	-	-	-	-	-	
171	18 096	28 281	576	1 989	-	X	-	-	X	11 185
173	56 860	77 264	17 090	55 568	-	-	-	-	-	
175	3 523	6 446	2 723	8 570	-	-	-	-	-	
177	5 066	9 112	3 704	11 797	-	-	-	-	-	
179	15 783	24 663	504	1 740	-	X	-	-	X	9 755
181	18 096	28 281	573	1 989	-	X	X	-	X	11 185
183	72 420	118 362	30 340	133 872	-	-	-	-	-	
185	3 523	6 446	2 723	8 568	-	-	-	-	-	
187	4 227	7 734	3 264	10 286	-	-	-	-	-	
189	7 400	13 600	5 802	18 212	-	-	-	-	-	
191	4 579	8 378	3 539	11 137	-	-	-	-	-	
193	7 030	12 920	5 510	17 294	-	-	-	-	-	
195	1 150	81 068	1 150	81 068	-	-	-	-	-	
197	1 534	127 011	1 534	127 011	-	-	-	-	-	
199	1 534	126 163	1 534	126 163	-	-	-	-	-	

instances. In particular, using local search to find an initial bound helps to solve large instances in which the initial kernelization step does not reduce the graph fully. Our full algorithm `FullA` can solve 82 out of the 100 instances, and in particular, as expeced, dominates each of the other configurations.

On the private instances, our full algorithm solved 87, the second place (peaty []) solved 77, the third place (bogdan []) solved 76 instances (of 100 instances). The peaty solver focused on using ..., whereas bogdan focussed on ...

6 Conclusion

We presented the winning solver of the PACE 2019 Implementation Challenge Vertex Cover Track. Our algorithm uses a portfolio of techniques, including an aggressive kernelization strategy with all known reduction rules, local search, branch-and-reduce, and a state-of-the-art branch-and-bound solver. Of particular interest is that several of our techniques were *not* from the literature on the vertex over problem: they were originally published to solve the (complementary) maximum independent set and maximum clique problems. Lastly, we performed extensive experiments to show the impact of the different solver techniques on the number of instances solved during the challenge. In particular, the results emphasize that data reductions play an important tool to boost the performance of the clique solver, and local search is highly effective to boost the performance of a branch-and-reduce solver for the independent set problem.

References

[1] Faisal N. Abu-Khzam, Michael R. Fellows, Michael A. Langston, and W. Henry Suters. Crown structures for vertex cover kernelization. *Theor. Comput. Syst.*, 41(3):411–430, 2007. `doi:10.1007/s00224-007-1328-0`.

[2] T. Akiba and Y. Iwata. Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover. *Theor. Comput. Sci.*, 609, Part 1:211–225, 2016. `doi:10.1016/j.tcs.2015.09.023`.

[3] D. V. Andrade, M. G.C. Resende, and R. F. Werneck. Fast local search for the maximum independent set problem. *Journal of Heuristics*, 18(4):525–547, 2012. `doi:10.1007/s10732-012-9196-4`.

[4] Luitpold Babel. A fast algorithm for the maximum weight clique problem. *Computing*, 52(1):31–38, 1994.

[5] Egon Balas and Chang Sung Yu. Finding a maximum clique in an arbitrary graph. *SIAM Journal on Computing*, 15(4):1054–1068, 1986.

[6] M. Batsyn, B. Goldengorin, E. Maslov, and P. Pardalos. Improvements to MCS algorithm for the maximum clique problem. *J. Comb. Optim.*, 27(2):397–416, 2014. `doi:10.1007/s10878-012-9592-6`.

[7] Shaowei Cai. Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[8] Shaowei Cai and Jinkun Lin. Fast solving maximum weight clique problem in massive graphs. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 568–574. AAAI Press, 2016. URL: `http://www.ijcai.org/Proceedings/16/Papers/087.pdf`.

[9] Lijun Chang, Wei Li, and Wenjie Zhang. Computing a near-maximum independent set in linear time by reducing-peeling. In *Proc. 2017 ACM International Conference on Management of Data (SIGMOD 2017)*, pages 1181–1196. ACM, 2017. `doi:10.1145/3035918.3035939`.

[10] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover: Further observations and further improvements. *Journal of Algorithms*, 41(2):280–301, 2001. `doi:10.1006/jagm.2001.1186`.

[11] Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. Accelerating local search for the maximum independent set problem. In *Proc. 15th International Symposium on Experimental Algorithms (SEA 2016)*, volume 9685 of *LNCS*, pages 118–133. Springer, 2016. `doi:10.1007/978-3-319-38851-9_9`.

[12] Y. Iwata, K. Oka, and Y. Yoshida. Linear-time FPT Algorithms via Network Flow. In *Proc. 25th ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 1749–1761. SIAM, 2014. URL: `http://dl.acm.org/citation.cfm?id=2634074.2634201`.

[13] C.E. Larson. A note on critical independence reductions. volume 51 of *Bulletin of the Institute of Combinatorics and its Applications*, pages 34–46, 2007.

[14] C.-M. Li, H. Jiang, and F. Manyà. On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. *Computers & Operations Research*, 84:1–15, 2017. `doi:10.1016/j.cor.2017.02.017`.

[15] G.L. Nemhauser and L. E. Trotter Jr. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8(1):232–248, 1975. `doi:10.1007/BF01580444`.

[16] Patric RJ Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1-3):197–207, 2002.

[17] Darren Strash. On the power of simple reductions for the maximum independent set problem. In *Proc. 22nd International Computing and Combinatorics Conference (COCOON 2016)*, volume 9797 of *LNCS*, pages 345–356. Springer, 2016. `doi:10.1007/978-3-319-42634-1_28`.

[18] Jeffrey S Warren and Illya V Hicks. Combinatorial branch-and-bound for the maximum weight independent set problem. 2006. URL: `https://www.caam.`

rice.edu/~ivhicks/jeff.rev.pdf.

[19] M. Xiao and H. Nagamochi. Confining sets and avoiding bottleneck cases: A simple maximum independent set algorithm in degree-3 graphs. *Theor. Comput. Sci.*, 469:92–104, 2013. doi:10.1016/j.tcs.2012.09.022.