WeGotYouCovered

Demian Hespe[1], Sebastian Lamm[2], Christian Schulz[3], and Darren Strash[4]

1 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
`hespe@kit.edu`
2 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
`lamm@kit.edu`
3 **University of Vienna, Faculty of Computer Science, Vienna, Austria**
`christian.schulz@univie.ac.at`
4 **Hamilton College, New York, USA,** `dstrash@hamilton.edu`

Abstract

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. For our submission, we use 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.

1998 ACM Subject Classification G.2.2 Graph Theory – Graph Algorithms, G.4 Mathematical Software – Algorithm Design and Analysis

Keywords and phrases kernelization, branch-and-reduce, local search

Digital Object Identifier 10.4230/LIPIcs.PACE.2019.

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., $\forall (u, v) \in E$ [$u \in S$ 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 [**?**], iterated local search for the maximum independent set problem [**?**], and a state-of-the-art branch-and-bound maximum clique solver [**?**].

We first briefly describe each of the techniques that we use, and then describe how we combine all of the techniques in our final solver.

2 Techniques

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 [**?**]. To compute a kernel, Akiba and Iwata [**?**] 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, unconfined vertex removal [**?**], a well-known linear-programming relaxation [**?**, **?**] related to crown removal [**?**], vertex folding [**?**], and twin, funnel, and desk reductions [**?**].

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.

Branch-and-Bound

Experiments by Strash [**?**] 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 [**?**], 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. [**?**], 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.

Iterated Local Search

Batsyn et al. [**?**] 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. [**?**] 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 [**?**, **?**].

3 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 [**?**]. 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 [**?**] 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 [**?**] 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.

4 Material

GitHub: `https://github.com/sebalamm/pace-2019/releases/tag/pace-2019`
DOI: `https://doi.org/10.5281/zenodo.2816116`