The DFVS-Via-Flattening Solver

Alex Meiburg

June 16, 2022

This document describes the algorithms in the PACE 2022 submission, "DVFS", available at https://doi.org/10.5281/zenodo.6650921, for solving exact minimal Directed Feedback Vertex Set (DFVS).

1 Approach

The central idea was: MILP solvers are very mature and can be adopted as a generic branch-and-bound framework; we should build our solver within this framework. SCIP, in particular, has a rich plugin system that allows custom behavior at each node, with custom heuristics for branch selection and heuristic solution generation, while still using the power of many other present reduction rules and heuristics. This vision of a cooperative SCIP-solver-plugin did not ultimately materialize, for a few reasons, including time constraints.

DFVS can be written down as an ILP in a simple form, with a cardinality constraint on each cycle in the graph. Since graphs can have exponentially many cycles, though, this does not give an efficient way of writing down the problem. One way is to do adaptive resolving: get an ILP using only some cycles, solve it to optimality, and then check the solution for any cycles still left in the graph. If there are, add them as new constraints and re-solve. If not, an optimum has been found. This is known as lazy reoptimization. Another approach is to add a callback in the ILP solver, so that when a node is first visited (but before optimality is reached), necessary cutting planes are added. This is known as a lazy constraint. This process can be greatly accelerated if a sufficient set of cycles is found before the solving.

Before reaching that point however, there many kernelizing reductions to be performed. The development effort focused on four problems then: efficient kernelizing rules for DFVS, finding a sufficient set of cycles, kernelizing on the cycle cover problem, and then optimizing the interaction with the ILP solver.

1.1 Kernelizing DFVS

After a few experiments, only a few kernelizing rules were used on the DFVS graph. It was found that the majority were not worth the (development) time, and could be more easily applied in the reduced cycle problem. The only rules were:

1. Split the graph into Strongly Connected Components (SCCs) and solve each separately.
2. Vertices with in-degree or out-dedgree one can be contracted with their unique neighbor.
3. Vertices with in- or out-degree zero can be removed.

All three rules are well known. Rules 2 and 3 cannot create new SCCs, so rule 1 only needs to be checked once. Rules 2 and 3 are efficiently implemented with a queue of "vertices to be checked" that ensures linear time kernelization, despite the fact that each can trigger new applications of the other.

1.2 Sufficient Cycles

Given a DFVS problem, we would like to find sufficient cycles, shrinking the graph as we go. In the vast majority of test problems, we were able to quickly find sufficient cycles and so could focus entirely on a minimal cover problem. The applied rules were:

1. Any K2 (vertices that pointed to each other) can be added as a cycle, then removed from the graph: as long as the cycle is covered, no other cycle can go through either edge.

2. A graph with at most one vertex with out-degree greater than 1 can be covered by the unique cycle running through each of the outgoing edges. There is also a complementary rule for at most one vertex with in-degree greater than 1.

3. Suppose there is a vertex u with edges to vertices v and w (possibly more). As long as v has out-degree one, keep finding its unique successor and calling that the new v. If this ever reaches w, then the whole path from $v \to u \to \dots w$ can be deleted.

4. Similar to above, but now $u \to v$ and $w \to u$ instead (the direction of the edge between w and u is reversed). Then the cycle from $v \to u \to \dots w \to v$ is added, and then the path $v \to u \to \dots w$ is deleted.

5. Split the graph into SCCs and find sufficient cycles for each. (This subsumes the rule that vertices with in- or out-degree zero can be removed.)

6. Given a weak articulation point (WAP) v, remove v and split into connected components C_i, then find sufficient cycles on the subgraph induced by each $C_i \cup \{v\}$.

7. Just as a WAP is a vertex whose deletion disconnects the graph, we can look for any edge $u \to v$ such that $G \setminus \{u, v\}$ is disconnected. Then each connected component C_i is analyzed on the subgraph induced by $C_i \cup \{u, v\}$.

8. If all the above fail to reduce the graph, then an exhaustive enumeration is attempted by traversing the graph in a depth-first search and looking for cycles. This is given a timeout in terms of number of vertices visited in the search. If this fails, a DFVS "chunk" remains.

This process returns a set of sufficient cycles, and possibly a few chunks. In most cases, no chunks were left.

1.3 Cycle Kernelization

Once sufficient cycles are found, we're left with a minimum cover problem: find the smallest number of elements (really, vertices) that cover the given sets (cycles). This simplified formulation has no notion of ordering in cycles or directedness, and greatly simplifies reduction rules. We applied several known kernelization rules from vertex cover literature, appropriately modified to handle set of size larger than 2. We think of the sets with two elements as "edges" (by analogy with vertex cover), and the rest as "big cycles". Each element has a "degree" equal to the number of edges containing it. We used reduction rules:

1. A vertex with degree 1 can be removed, unless it has a big cycle. Its neighbor is included in the cover.

2. A vertex with degree 2 and no big cycles can be either dropped (if it belongs to a K3) and all its neighbors included, or it can be folded (its neighbors do not neighbor one another).

3. If a vertex has no big cycles and has a simplicial neighborhood, it can be dropped and all its neighbors included.

4. If a vertex has no big cycles and dominates another vertex, include the vertex.

5. If a vertex is part of a "funnel" and not a big cycle, it can be folded.

6. The "confining set" of a vertex can be computed, and if it is unconfined and nothing in the set $N(S)$ has a big cycle, the vertex can be included.

7. If a big cycle contains another big cycle or an edge, it is redundant and the larger cycle can be removed.

8. If a vertex has degree zero and one big cycle, the vertex can be dropped.

During the above processing, a big cycle can shrink only due to the last rule. If it shrinks to size 2, it becomes an edge. Although seveal of the first rules could be modified to included cases where more vertices do belong to big cycles, these were found to be almost entirely subsumed by the first cases.

When chunks were present, vertices belonging to chunks were never dropped; sometimes, however, vertices could be safely included. This triggered further some kernelization of the graph, but not new cycle-finding. This is a room for future improvement.

When no big cycles or chunks were present, and it was a pure VC problem, "desk" reductions were also employed.

1.4 MILP Interaction

When chunks remained present, the problem was still not in the form of a MILP, and had to be lazily enforced. A great deal of effort went into finding the best way to interact with SCIP in the most productive and cooperative way, including lazy constraint enforcement at nodes and providing heuristic solutions as upper bounds. Ultimately these were unproductive, because the very intelligent "presolver" in SCIP did not have complete information on the problem – it knew that there were lazy constraints not expressed by the ILP – and so could not do many of its most useful tricks. The simplest loop of "solve to optimum, find new cycles" turned out to be the most efficient. Significant improvement was found when an initial random search through the graph to just find *some* large list of cycles as a starting point; however, too many initial cycles lead to performance degradation, presumably due to SCIP processing the very large number of constraints.

Generated cycles were found through a randomized DFS exploration until returning to a previous node. The found cycle would then be trimmed at any chords until it was chordless (minimal). Finally, a random edge from the cycle would be removed from (the temporary copy of) the graph to prevent it being found again in this round, and the new a new cycle was sought. This continued until the temporary copy of the graph was acyclic. This would then be executed 50 times with different random seeds. This happened once before the first MILP solving, and once after each candidate optimum was found.

When no chunks were present, using SCIP efficiently was easier. Setting the presolver to "aggressive" and the emphasis to "optimality" produced the best peformance. As SCIP offers many different built-in constraint types, we had the choice of adding the constraint as simple "linear" constraints, "logical or" constraints, or "packing" constraints. It was found best to use "linear" constraints, and the presolver then "upgraded" some of them appropriately to the other two as needed. An experiment was run with using the negations of all the variables, and it was found to decrease performance; this suggests it would be useful to negate, then, if it was instead a maximum independent set problem.