Definitions

The set of vertices adjacent to vertex v (its neighborhood) is denoted by $N(v)$. The closed neighborhood of v is denoted by $N[v] = N(v) \cup \{v\}$.

Method

The solver assumes an instance to contain the graph $G = (V, E)$ itself as well as:

- A set of vertices $D \subseteq V$ already a part of the solution.
- A set of vertices $X \subseteq V$ where a proven optimal solution to the instance exists where no vertex in X is part of the solution.
- A set of vertices $W \subseteq V$ that do not need to be dominated. E.g. as a result of being in the neighborhood of a vertex in D.

The solver implements a classic branch and bound procedure using the branching heuristic desribed in [**bab**] - only with ties in the ordering of dominators of the chosen vertex broken by preferring vertices closest to 0.5-assignments by the lower bound procedure (intuitively most undecided).

As in [**bab**] we terminate a given branch when we determine its lower bound to match (or lose to) the currently most optimal solution: no further recursion will yield a more optimal solution. The lower bound method is implemented using the same linear program formulation as in [**bab**] where the set I corresponds to our set W and the set S corresponds to our set D.

We also terminate the current branch if $200 < |V \setminus D| < 400$. In this case we solve the same linear program as mentioned above, only where an integer solution is enforced, effectively solving the remaining undecided part of the current instance. The conditions under which to take this path, i.e. $200 < |V \setminus D| < 400$, were found experimentally.

Lastly, the branch and bound implementation performs combinatorical reductions on the given instance, transforming it into an easier to solve instance (fewer undecided choices to make) that also remains an optimal substructure to the original instance.

Reduction rules

The three rules described in [**bab**] have been used in this solver as well, only with a few minor details in their implementations aimed at increasing efficiency. In particular:

- "Rule 2" from [**bab**]: For all undetermined vertices $u \bigcap_{w \in C(u)} N[w]$, if this intersection is non-empty, the property is upheld for 2 undetermined edges.
- "Rule 3" from [**bab**]: Similarly as for rule 2, for all undominated vertices $u \bigcap_{w \in D(u)} N[w]$, then all elements in the intersection would have the property together with u.

The two rules described in [**rule12**] were also considered:

- "Rule 1" from [**rule12**] This is a special-case of the combination of rules from [**bab**].
- "Rule 2" from [**rule12**] This rule was implemented as part of the reduction step.

A few simpler rules were implemented as well. For these, efficiency was a main concern; the goal was to cheaply make trivial reductions to reduce the number of necessary iterations of the more complex rules:

- If all remaining vertices are in W, erase all of them: the instance is solved, erase all remaining vertices so as to avoid spending time on other rules.
- If $N[v] \cap D \neq \emptyset$ and $v \in X$ for some $v \in V$, then remove v from V: v is already dominated and cannot dominate others as per $v \in X$.
- If $N(v) = \emptyset$, if $v \notin D$, insert v into D, then remove v: v has no influence on the rest of G.
- If $N[v] \cap D \neq \emptyset$ and $N[u] \cap D \neq \emptyset$ and $(u, v) \in E$, remove (u, v) from E: we do not need to dominate v using u and vice-versa.

- Define a *triangle* to be a construction of three vertices u, v, w where $N(v) = \{u, v\}$ and $(u, v) \in E$ and $w \notin W \cup D$. If we have two such triangles $\{u_1, v_1, w_1\}$ and $\{u_2, v_2, w_2\}$, remove (if present) the edges $(u_1, u_2), (u_1, v_2), (v_1, u_2), (v_1, v_2)$: it is clear that none of w_1, w_2 need to be in D as if for some triangle $\{u, v, w\}$ w were in D then instead retain an optimal substructure by dominating w using either u or v. If we end up inserting u into D, then v is dominated by the presence of $(u, v) \in E$ (and vise-versa). Thus, any triangle will not need any outside vertices to dominate any of u, v, w. So like in the rule where we erase edges between two dominated vertices, erase these four edges as well.

Additionally, we recursively solve disjoint components by themselves.

Lastly, we split the graph around articulations as well as *2-cuts* (vertex-cuts of size two) in a way that avoids branching by solving each side of disconnection separately:

Articulation-point rule

Represent a minimum and feasible soltuion to some instance G as $T(G)$. Assume all vertices in D are erased from the instance.

Consider some $v \in V$ where v is an articulation point and consider some $C \subseteq V \setminus \{v\}$ such that the induced graph $G[C]$ is connected and $\nexists C' \supsetneq C, C' \cap \{v\} = \emptyset$ such that $G[C']$ is connected. Let $c_{v \in D} = |T(G[C \cup \{v\}])| - 1$, i.e. the cost of solving C where $C \cap N(v)$ is already dominated for free, and let $c_{v \notin D} = |T(G[C])|$. We may reduce G by:

- **Case 1.** If $c_{v \notin D} < c_{v \in D}$: This is impossible as any solution to $G[C]$ where $C \cap N(v)$ is not already dominated also will be a solution where $C \cap N(v)$ already is dominated.
- **Case 2.** If $c_{v \notin D} = c_{v \in D}$: There are two sub-cases here:
  - **a.** $\exists T(C) : N(v) \cap T(C) \neq \emptyset \wedge |T_C| = |T(G[C])|$, then set $D := D \cup T_C$: prefer a solution that also dominates v.
  - **b.** otherwise use any solution $T(G[C])$ and set $D := D \cup T(G[C])$.
- **Case 3.** This implies that no solution to C exists that beats one where $C \cap N(v)$ already is dominated, even when paying for the inclusion of v in D. Therefore, pick any solution $T(G[C])$ where $C \cap N(v)$ already is dominated, set $D := D \cup T(G[C]) \cup \{v\}$.

This way, we effectively erase all but one component divided by v by solving those respective components, in isolation, at most three times apiece. We leave only the largest component in order to avoid solving that more than once.

2-cut rule

As in the articulation-point rule, the goal is to erase a component removed from the rest of G due to some vertex-cut $\{u, v\}$ by solving it some number of times in isolation.

Assume all vertices in D are erased from the instance.

Consider some $u, v \in V$ where $\{u, v\}$ is a vertex-cut and consider some $C \subseteq V \setminus \{u, v\}$ such that the induced graph $G[C]$ is connected and $\nexists C' \supsetneq C, C' \cap \{u, v\} = \emptyset$ such that $G[C']$ is connected. Compute for each assignment of decisions on u and v the cost of solving $G[C]$. The assignment decisions on one of u and v are:

1. Either $\in D$ or
2. $\notin D$ but dominated by C or
3. $\notin D$ but dominated externally.

There are 9 assignments of u and v together.

If we can find a "gadget" graph with the same relative costs as C under those 9 assignments, then we can replace C by that gadget, solve the new instance, and then replace the gadget with the appropriate solution to C (i.e. the one that matches the actual final assignments of u and v). This is efficient when the gadget is not too large.

Using a brute force implementation in `CUDA` we were able to find nearly all possible gadgets. We searched only in graphs of size $1 \ldots 7$ (excluding the two cut-vertices). We clamped the relative costs (which identifies a given gadget) to between 0 and 2 (inclusive) after normalizing with respect to the cheapest of the 9 solutions since any

solution more expensive than 2 compared to the cheapest is worse than simply using that cheapest solution as well as paying for inserting u and v into D.

Splitting the graph (e.g. solving C by itself) like this introduces issues due to possible dependencies between vertices on either side of the cut now being destroyed. In the articulation-point rule, this is handled trivially in the implementation. For 2-cuts it was solved by removing all labels from vertices in V. Experimenting showed that this rule was a major speed-up even when suffering this compromise due to all labels being re-assigned in a single pass of reductions on the given modified graph.

Special case: vertex cover

We noticed that several of the cases in the challenge encoded vertex cover instances. In these cases we built the corresponding vertex cover instance and solved it using [**pace19**].

Claim. If $V = X \cup Y$ where $X \cap Y = \emptyset$. Then if for every node $x \in X, |N(x)| = 2, N(x) \subseteq Y, N[x]$ forms a clique and $\bigcup_{x \in X} N(x) = Y$. Then a minimum vertex cover of the induced graph $G[Y]$ is a minimum dominating set of G.

Proof. First notice that if $x \in X$ is in the dominating set, we can remove x from the dominating set and insert one of the neighbors of x. Then it will continue being a dominating set since x and $N(x)$ will be dominated by the neighbor since $N[x]$ forms a clique. This will be a dominating set of equivalent size. Therefore, there exists a minimum dominating set which is a subset of Y. By the construction of the graph $G[Y]$ we notice that all edges in $G[Y]$ implies that one of the 2 nodes should be in the dominating set since they need to cover the respective $x \in X$. The minimum set to approve all the constraints is equivalent to the minimum vertex cover of $G[Y]$.

LP solver

We used the `COIN-OR Cbc` [**coin**] LP solver which is also capable of solving MLP instances.