Appendix

This appendix was meant to develop some aspects of the work "A note on the Maximum Weighted Irredundant Set".

A word about the formalization of graph theory. The *Formalization of Mathematics* is an area of Computer Science that expresses and proves mathematical statements in a highly structured language using a pre-established set of inference rules. Correctness of results is automatically checked by a tool called *proof assistant*. Formalizing proofs provides several benefits including new ways of visualizing and interacting with the mathematical corpus[1].

Currently, there is a community devoted to formalizing known results of several branches of Mathematics. In the case of graph theory, one of the longest proofs ever formalized in Coq is the four color theorem[2]. We recall that this theorem was first proved by K. Appel and W. Haken and it had the peculiarity that it was necessary to prove thousands of cases, called *configurations*, although this task could be carried out mechanically by a computer. Later, the number of configurations was reduced to 633 but it is still a large number to be considered "manually verifiable". The formalization of such a result in a proof assistant such as Coq has the advantage of reducing trust only to the proof assistant, without involving other software. That is, a "skeptical" reader neither needs to trust in a program that checks all the configurations nor has to make her/his own program to convince herself/himself of the correctness of the theorem, she/he only needs to trust in the proof assistant.

Our proposal is that long proofs (involving a large number of cases that can be checked mechanically) can be channeled through a proof assistant. As libraries of formalized mathematics are mature enough, we encourage other researchers to use this mechanism to present such proofs so that a reader is not subjected to check a large number of cases to be sure of the veracity of a result.

We also explored the possibility of generating a Coq file that *certificates* a numerical value for a given graph parameter (in our case, IR_w). This idea came to our mind after knowing many works about optimization problems in graphs where the authors give, for example, a new bound of some parameter that improves the existing ones, but one has to trust what the author claims. We discuss it at the end of this appendix.

Selection of instances. They were taken from `https://mat.gsia.cmu.edu/COLOR04`. It is a standard set of instances which were originally chosen for benchmarking graph coloring algorithms, although later they were used for other optimization problems in graphs, in particular for dominating set prob-

[1]Harrison J., *Formal proof - Theory and Practice*, Notices Amer. Math. Soc. **55** (2008), 1395–1406.

[2]Gonthier G., *Formal proof - the Four-Color Theorem*, Notices Amer. Math. Soc. **55** (2008), 1382–1393.

lems[3]. In addition, a weighted instance with data taken from `https://www.buenosaires.gob.ar/laciudad/barrios` was considered. In this instance, the vertices of the graph and their weights are respectively the districts of the city of Buenos Aires and their population, and there is an edge between two vertices if the corresponding districts are neighbors. The following table shows the name of these districts and their population in thousands. Those districts that belong to the irredundant set of maximum weight are highlighted. As this set is dominating, it is also an optimal solution of the WUDS problem (i.e., $IR_w(G) = \Gamma_w(G) = 1634$).

Name	Population	Name	Population
Agronomía	35	Parque Chas	39
Almagro	139	Parque Patricios	41
Balvanera	152	Puerto Madero	7
Barracas	77	Recoleta	189
Belgrano	139	Retiro	45
Boedo	49	Saavedra	52
Caballito	183	San Cristóbal	50
Chacarita	27	San Nicolás	33
Coghlan	19	San Telmo	26
Colegiales	57	Vélez Sarsfield	36
Constitución	46	Versalles	14
Flores	150	Villa Crespo	90
Floresta	39	Villa del Parque	59
La Boca	46	Villa Devoto	71
La Paternal	20	Villa Gral. Mitre	36
Liniers	44	Villa Lugano	114
Mataderos	65	Villa Luro	33
Montserrat	44	Villa Ortúzar	23
Monte Castro	35	Villa Pueyrredón	40
Nueva Pompeya	63	Villa Real	14
Núñez	53	Villa Riachuelo	15
Palermo	252	Villa Santa Rita	34
Parque Avellaneda	54	Villa Soldati	41
Parque Chacabuco	39	Villa Urquiza	89

Experiment on larger instances. We generated weighted random instances with different edge densities (the procedure starts with an edge-less graph of $n \in \{250, 500, 1000\}$ vertices and adds edges with a given probability $p \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$; for computing weights, numbers from $\{1, 2, 3, 4, 5\}$ are taken with a uniform distribution). For each instance, we run our heuristic and a greedy heuristic that tries to find a stable set of maximum weight of the transformed graph G' of Theorem 3.1. We also run both heuristics on the real instance. The greedy heuristic is given as follows.

[3]Chalupa D., *An order-based algorithm for minimum dominating set with application in graph mining*, Inf. Sci. **426** (2018), 101–116.

Start with $W \leftarrow V$ and $S_{max} \leftarrow \emptyset$. Then, repeat the following until W is empty:

- Pick $v \in W$ such that $w(v)$ is highest. In case of a tie, among those vertices v having the same weight, pick the one that minimizes $|N_{G'}(v) \cap W|$.
- Update $S_{max} \leftarrow S_{max} \cup \{v\}$ and $W \leftarrow W \setminus N_{G'}[v]$.

where $N_{G'}(v)$ denotes the set of neighbors of v in G' and $N_{G'}[v] \doteq N_{G'}(v) \cup \{v\}$. After the loop, S_{max} has the resulting stable set of G'.

The table given below reports the results. The best values are highlighted in boldface. The symbol "–" means that the algorithm runs out of memory.

			heuristic		greedy heuristic		
Name	$\vert V(G)\vert$	dens G	$w(D_{max})$	Time (sec.)	$\vert V(G')\vert$	$w(S_{max})$	Time (sec.)
R250_10	250	10	**184**	13	6590	157	**0.28**
R250_30	250	30	**85**	7.18	18988	69	**1.73**
R250_50	250	50	**53**	5.37	31520	40	**4.99**
R250_70	250	70	**35**	**3.63**	43840	25	9.45
R250_90	250	90	**22**	**1.93**	56242	15	19.22
R500_10	500	10	**236**	186	25130	188	**4.5**
R500_30	500	30	**102**	103	75322	82	**61**
R500_50	500	50	**61**	**78**	126016	49	229
R500_70	500	70	**40**	**57**	175202	–	–
R500_90	500	90	**25**	**33**	225006	–	–
R1000_10	1000	10	**284**	2586	101208	243	**136**
R1000_30	1000	30	**113**	**1491**	300518	–	–
R1000_50	1000	50	**69**	**1227**	500294	–	–
R1000_70	1000	70	**45**	**940**	700542	–	–
R1000_90	1000	90	**27**	**575**	899186	–	–
buenosaires	48	10	**1634**	0.01	278	1340	<**0.01**

As one can see in the table, our heuristic delivers solutions of better quality in all the tested instances. The greedy heuristic is faster than ours for low-density graphs; however, for graphs of higher densities, it is unable to allocate the adjacency matrix of G' (in 16Gb of available memory). We expect that, even changing the representation of G' in memory, this approach will not scale for larger instances.

Sketch of the proofs of the necessity parts of Lemmas 3.2 and 3.3. Suppose that G' has a claw as induced subgraph and we want to prove that G is not $\{\text{claw}, \text{bull}, P_6, \overline{C_6}\}$-free. The rationale is to divide the proof into several cases in each of which we can find a claw, a bull, a P_6 or a $\overline{C_6}$ in G.

We recall that a claw is a graph of 4 vertices such that one (called *central*) is adjacent to the other three. As claw $\dot{\subset}$ G', there is an injective map $f' : V(\text{claw}) \to V'$ that preserves adjacencies. If $a_1a_2, b_1b_2, c_1c_2, d_1d_2 \in V'$ with a_1a_2 adjacent to b_1b_2, c_1c_2 and d_1d_2 in G', are the images by f' of the vertices of the claw, then we know the following[4]:

A. *Definition of* V'. For each $x \in \{a, b, c, d\}$, since $x_1x_2 \in V'$, we have $x_1 = x_2$ or x_1 is adjacent to x_2 in G.

[4]In the Coq file (`mwis_prop.v`) the names of each of these condition are: A. `a1a2`, `b1b2`, `c1c2`, `d1d2`; B. `a1a2b1b2`, `a1a2c1c2`, `a1a2d1d2`, `b1b2c1c2`, `b1b2d1d2`, `c1c2d1d2`; C. `a1b2a2b1`, `a1c2a2c1`, `a1d2a2d1`; D. `b1c2b2c1`, `b1d2b2d1`, `c1d2c2d1`.

B. *Injectivity of f'.* For any $x, y \in \{a, b, c, d\}$ such that $x \neq y$, we have $x_1x_2 \neq y_1y_2$. That is, $x_1 \neq y_1$ or $x_2 \neq y_2$.
C. *Preservation of adjacencies.* For each $x \in \{b, c, d\}$, since a_1a_2 is adjacent to x_1x_2 in G', we have $a_1 = x_2$ or a_1 is adjacent to x_2 in G or $a_2 = x_1$ or a_2 is adjacent to x_1 in G.
D. *Preservation of no-adjacencies.* For any $x, y \in \{b, c, d\}$ such that $x \neq y$, since x_1x_2 is not adjacent to y_1y_2 in G', we have $x_1 \neq y_2$, x_1 is not adjacent to y_2 in G, $x_2 \neq y_1$ and x_2 is not adjacent to y_1 in G.

The simplest case is when $a_1 = a_2$, $b_1 = b_2$, $c_1 = c_2$ and $d_1 = d_2$. In this case, we exhibit a claw in G, i.e., we provide an injective map $f : V(\text{claw}) \to V$ that preserves adjacencies. Let f map the central vertex to a_1 and the others to b_1, c_1 and d_1. By condition B, the vertices a_1, b_1, c_1 and d_1 are different in G, so f is injective. By conditions C and D, a_1 is adjacent to b_1, c_1 and d_1 but b_1, c_1 and d_1 are not adjacent to each other, thus f preserves adjacencies. The next case is when $a_1 = a_2$, $b_1 = b_2$, $c_1 = c_2$ and $d_1 \neq d_2$. Again, we exhibit a claw in G by proposing that f maps the central vertex to a, two of the other three vertices to b and c, and the remaining one to d_1 if a_1 is adjacent to d_1, or d_2 otherwise. We proceed as before and use the conditions above to prove that f is injective and preserves adjacencies. Note that, by symmetry, we also proved the case $a_1 = a_2$, $b_1 \neq b_2$, $c_1 = c_2$, $d_1 = d_2$, and the case $a_1 = a_2$, $b_1 = b_2$, $c_1 \neq c_2$, $d_1 = d_2$. This is how the proof is systematically constructed.

Proof of Lemma 3.5. Since $H \dot{\subset} G$, there exists an injective map $f : V(H) \to V(G)$ that preserves the edge relationship. Let $f' : V(H') \to V(G')$ be the map defined by $f'(uv) = f(u)f(v)$. Note that the injectivity of f' follows readily from the injectivity of f. So to prove the lemma, it suffices to show that f' preserves the edge relationship. Indeed, we have $(uv, zr) \in E(H') \Leftrightarrow ((uv \neq zr) \wedge (v \in N[z] \vee r \in N[u])) \Leftrightarrow ((f(u)f(v) \neq f(z)f(r)) \wedge (f(v) \in N[f(z)] \vee f(r) \in N[f(u)])) \Leftrightarrow (f'(uv), f'(zr)) \in E(G')$, where the second equivalence follows from the fact that f is injective and preserves the edge relationship. □

Proof of Corollary 3.6. By Lemma 3.5, we have $H' \dot{\subset} G'$, and so $K \dot{\subset} G'$ by the transitivity of the induced subgraph relation. □

Proof of Lemma 3.7. Let $f : V \to V'$ be the map defined by $f(v) = vv$. Since f is clearly injective, to prove the lemma it is enough to show that it preserves the edge relationship. Indeed, we have $(u, v) \in E \Leftrightarrow ((u \neq v) \wedge (u \in N[v])) \Leftrightarrow ((uu \neq vv) \wedge (u \in N[v] \vee v \in N[u])) \Leftrightarrow (uu, vv) \in E'$, which completes the proof. □

Proof of the fact that $\mathcal{Q}$ is a set of cliques that covers all the edges of G'. In Section 4, we proposed

$$\mathcal{Q} = \big\{\{uv : v \in N[u] \cap N[z]\} \cup \{zr : r \in N[z]\} : u, z \in V \text{ such that } u \neq z,\ N[u] \cap N[z] \neq \emptyset\big\}.$$

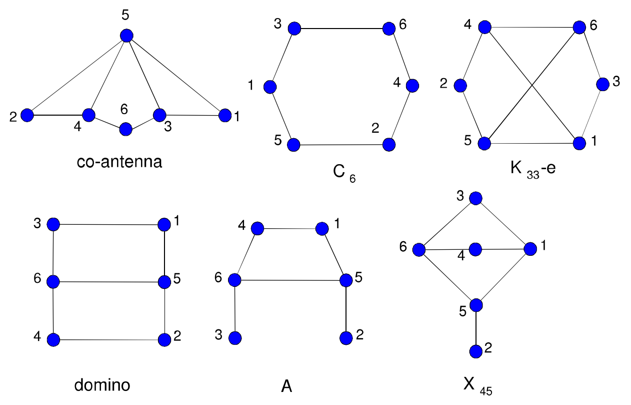

Figure 4: A co-antenna, C_6, $K_{3,3}$-e, domino, A, and X_{45}.

as a set of cliques that covers all the edges of G'. In fact, let u, z be different vertices such that $N[u] \cap N[z] \neq \emptyset$, and define $Q^1_{uz} \doteq \{uv : v \in N[u] \cap N[z]\}$ and $Q^2_z \doteq \{zr : r \in N[z]\}$. Clearly, Q^1_{uz} and Q^2_z are cliques of G'. Moreover, $Q^1_{uz} \cup Q^2_z$ is also a clique of G' since any $uv \in Q^1_{uz}$ is adjacent to any $zr \in Q^2_z$ (due to $v \in N[z]$). On the other hand, for any $e = (ab, cd) \in E'$, we have w.l.o.g. that $b \in N[c]$. Since $ab \in V'$, also $b \in N[a]$. If $a = c$, then let $u \in N(c)$. Hence, $Q^1_{uc} \cup Q^2_c$ covers e. Otherwise, $a \neq c$ and therefore $Q^1_{ac} \cup Q^2_c$ covers e.

Proof of the fact that if G has a C_5, C_6, $K_{3,3}$-e, domino, co-antenna, A, or X_{45} then G' has a C_5. In Section 4, we mention that the previous statement (if G has certain subgraphs, then G' has a C_5) can be proven. By Corollary 3.6, it is enough to prove that, for any $H \in \{C_5, C_6, K_{3,3}\text{-e}, \text{domino}, \text{co-antenna}, A, X_{45}\}$, we have $C_5 \dot{\subset} H'$. The fact that $C_5 \dot{\subset} C'_5$ follows from Lemma 3.7. If G has a C_6 with the vertices labeled as in Figure 4, then the vertices 11, 25, 42, 66 and 63 induce a C_5 in G'. We proceed as above with the remaining subgraphs by exhibiting those vertices that induce a C_5 in G':

- $K_{3,3}$-e: 22, 51, 31, 63, 64.
- domino: 11, 51, 22, 64, 63.
- co-antenna: 11, 25, 42, 66, 63.
- A: 11, 14, 63, 36, 52.
- X_{45}: 33, 41, 52, 25, 63.

More about the formalization of graph theory. One of the most established proof assistants, and with a large community, is *Coq*[5]. It works with the

[5]see `https://coq.inria.fr`

theory of *Calculus of Inductive Constructions*. In our case, we use Coq with an extension called *Ssreflect*[6]. This extension, together with the *Mathematical Components*[7] (a vast library of formalized mathematical results), was used to prove the four color theorem. A comprehensive introduction is given in the Handbook of Mathematical Components[8].

In the case of graph theory, besides the four color theorem, there is a graph library in Coq/Ssreflect with the definitions of simple graphs, digraphs, multigraphs, paths, trees, among much others concepts, and various results such as Menger's theorem, the characterization of graphs with treewidth 2 and the Cockayne-Hedetniemi domination chain[9]. One of the latest developments is the formalization of the Lovász replication lemma and the weak perfect graph theorem[10].

We next present a very brief introduction to Coq and its type theory; a reader who is already familiar with this language can skip this part. In Coq, every element `a` has a type `A`, denoted by `a : A`, and this relationship can be considered somehow a set membership, i.e., `a` $\in$ `A`. When `a : A`, it is said that `a` is an *inhabitant* of `A`. For instance, `nat` is the type of non-negative integer numbers and $0, 1, 2, \ldots$ are its inhabitans. There are type operators that allows one to construct more complex types, one of them is $\to$ as in standard mathematics, e.g., `f : A → B` is a function that maps elements of type `A` to others of type `B`, and `g : A → B → C` is a function that takes arguments of types `A` and `B`, and returns an element of type `C`[11].

A feature of Coq is that a type can *depend* on other types; the graph constructor `SGraph` (defined in the file `sgraph.v` of the graph library) is an example of that:

```
SGraph : ∀ (svertex : finType) (sedge : rel svertex),
         symmetric sedge → irreflexive sedge → sgraph
```

which means that `SGraph` generates a simple graph (whose type is `sgraph`) from:

[6] Gonthier G. and A. Mahboubi, *An introduction to small scale reflection in Coq*, J. Form. Reason. **3** (2010), 95–152.

[7] see `https://github.com/math-comp/math-comp`

[8] see `https://math-comp.github.io/mcb`

[9] see `https://github.com/coq-community/graph-theory`, also see the following works: Doczkal C. and Pous D., *Graph Theory in Coq: Minors, Treewidth, and Isomorphisms*, J. Autom. Reasoning **64**, 795–825 (2020); Doczkal C. and Pous D., *Completeness of an axiomatization of graph isomorphism via graph rewriting in Coq*, Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs **197**, 325–337 (2020); Severín D., *Formalization of the Domination Chain with Weighted Parameters*, Lebniz. Int. Proc. Inform. **141**, 36:1–36:7 (2019).

[10] Singh A., and R. Natarajan, *A Constructive Formalization of the Weak Perfect Graph Theorem*, Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs **197** (2020), 313–324.

[11] As $\to$ associates to the right, `g` indeed takes an argument of type `A` and returns a function of type `B` $\to$ `C`, which subsequently maps elements from `B` to `C`, giving effect to a two-argument function when it is evaluated.

a set of vertices `svertex`, a relationship `sedge` among vertices and "proofs" that `sedge` is symmetric and irreflexive (so that `sedge` becomes an edge relationship). Note how `sedge` is defined in terms of `svertex`, exposing a dependent type.

The type of mathematical statements (or *propositions*) is called `Prop`, an example was given above: "`symmetric sedge : Prop`". To illustrate better this concept, let `A` $\doteq$ `∀ n : nat,∃ m : nat, m = n+1`, then `A : Prop`. The previous expression `A` is itself a type, and any proof of `A` is an inhabitant of it. Thus, to prove that `A` is true, one has to propose a certain object `a : A`, which is constructed from previous declarations and statements via *tactics*, i.e., commands from a specific language that help to construct proofs. An example taken from the file `dom.v` of the graph library, which states that the empty set is stable, is:

```
Lemma stable0 : stable ∅.
Proof.
   by apply/stableP=> ? ?; rewrite in_set0.
Qed.
```

Here, the new statement `stable0` is declared and its proof is given between the commands `Proof` and `Qed`, with the use of the tactics `apply` and `rewrite` (they are described in the Handbook of Mathematical Components), and the previously defined objects `stableP` and `in_set0` (e.g., the latter states that $x \notin \emptyset$ for any x). If Coq parses succesfully these lines of codes, the statement `stable0` is true (the empty set **is** stable) and it is available to other further results.

Another useful type is `bool`, with only two inhabitants: `true` and `false`. When using the extension *Ssreflect*, a key concept is the *boolean reflection*: a correspondence between propositions and boolean objects. This correspondence allows to prove a proposition by manipulating a boolean object and viceversa (for instance, one may use the Axiom of Choice on the boolean side, since it is not available naturally in the Coq logic), and is declared with `reflect` through the so called *reflection lemmas*. Below is another example taken from `dom.v`, where it is stated that the definition of a dominating set `D`:

```
(∀v ∉ D → ∃ u ∈ D ∧ u -- v) : Prop
```

is equivalent to the object:

```
[forall (v | v \notin D), exists u in D, u -- v] : bool
```

that reduces to `true` if and only if `D` is dominating:

```
Lemma dominatingP_alt :
   reflect (∀v ∉ D → ∃ u ∈ D ∧ u -- v)
           [forall (v | v \notin D), exists u in D, u -- v].
```

where the symbol "`--`" denotes the edge relationship. Depending on the circumstances, one may use the `Prop` version or the `bool` version of the notion of dominating set.

Additional material related to this work. In the GitHub repository `https://github.com/aureus123/graph-theory/tree/mwis`, one can find:

- The folder `mwis` containing the theory developed in Sections 2 and 3 (split up into the files `prelim.v`, `mwis.v` and `mwis_prop.v`).
- The file `check_ir.v` (also in the folder `mwis`) containing auxiliary functions for the certificates generated by the solver.
- The folder `solver` with the implementation of the heuristic, the integer linear formulations for the MWIS and WUDS problems and the generator of Coq certificates.
- The folder `instances` with the instances used in the experiments.
- The folder `certs` with the certificates of the evaluated instances.
- This appendix.

Below, we make a brief description of the relevant objects contained in the files `prelim.v`, `mwis.v` and `mwis_prop.v` (which are in the folder `mwis`):

- `induced_hom`: given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ and a map $h : V_1 \to V_2$, it defines that h is an *induced homomorphism* if it satisfies that $(x, y) \in E_1 \Leftrightarrow (h(x), h(y)) \in E_2$ for all $x, y \in V_1$.
- `induced_subgraph`: for given graphs G_1 and G_2, it defines that G_1 is an *induced subgraph* of G_2 ($G_1 \dot{\subset} G_2$) if there exists an injective induced homomorphism from G_1 to G_2.
- `subgraph_trans`: it establishes that, if $G_1 \dot{\subset} G_2$ and $G_2 \dot{\subset} G_3$, then $G_1 \dot{\subset} G_3$.
- `trfgraph`: it is a function that constructs the graph G' of Theorem 3.1 from a given graph G.
- `trfgraph_subgraph`: given two graphs G and H such that G is an induced subgraph of H, it establishes that G' is an induced subgraph of H' (Lemma 3.5).
- `subgraph_G_G'`: it establishes that $G \dot{\subset} G'$ (Lemma 3.7).
- `irred_G_to_stable_G'`: given an irredundant set $D \subset V(G)$, it establishes that there is a stable set $S \subset V(G')$ of the same weight as D.
- `stable_G'_to_irred_G`: given a stable set $S \subset V(G')$, it establishes that there is an irredundant set $D \subset V(G)$ of the same weight as S.
- `IR_w_G_is_alpha_w_G'`: it establishes that $IR_w(G) = \alpha_{w'}(G')$ (Theorem 3.1).

- `IR_w_leq_V_minus_delta_w'`: it establishes that $IR_w(G) \leq w(V(G)) - \delta_w(G)$ (Lemma 2.1).
- `Knm`: it is a function that returns the complete bipartite graph $K_{n,m}$ for any $n, m \in \mathbb{N}$.
- `Pn`: it is a function that returns the path graph P_n for any $n \in \mathbb{N}$.
- `Cn`: it is a function that returns the cycle graph C_n for any $n \in \mathbb{N}$.
- `CCn`: it is a function that returns the complement of C_n for any $n \in \mathbb{N}$.
- `claw`: it is the graph $K_{1,3}$.
- `bull`, `G7_1` and `G7_2`: they are the last three graphs of Figure 2.
- `copaw`: it is the complement of a paw (for the latter, see Figure 2).
- `copaw_sub_copaw'` and `claw_sub_claw'`: they prove that co-paw and claw are induced subgraphs of co-paw′ and claw′ respectively, which are direct consequences of Lemma 3.7:

  ```
  Lemma copaw_sub_copaw' : copaw ⊂̇ trfgraph copaw.
  Proof. exact: subgraph_G_G'. Qed.
  ```

- `copaw_sub_G7_1'`, `copaw_sub_G7_2'`, `claw_sub_bull'`, `claw_sub_P6'` and `claw_sub_CC6'`: they are proofs of the facts represented in Figure 3.
- `G'clawfree` and `G'copawfree`: they establish the sufficiency parts of Lemmas 3.2 and 3.3.
- `G'clawfree_rev` and `G'copawfree_rev`: they establish the necessity parts of Lemmas 3.2 and 3.3.
- `clawfree_char` and `copawfree_char`: generalized versions of Lemmas 3.2 and 3.3. For instance, in the latter case, given graphs `Gcopaw` (isomorphic to co-paw), `GG7_1` (isomorphic to G^7_1) and `GG7_2` (isomorphic to G^7_2), it states that `Gcopaw`$\dot{\subset} G$ or `GG7_1`$\dot{\subset} G$ or `GG7_2`$\dot{\subset} G$ if and only if `Gcopaw`$\dot{\subset} G'$.

About the generation of certificates. One of the benefits of having the theory formalized in a proof assistant is that, with little additional effort, one can provide a file with a certificate of a certain value or bound of a parameter for a given instance; in our case, it is a lower bound of $IR_w(G)$ obtained by exhibiting an irredundant set whose weight is that bound. Moreover, it is irrelevant how the irredundant set was found, it can be a black box (e.g., CPLEX) or even a buggy code. As long as Coq can parse the file succesfully, one can be sure that the given bound is valid. No other software (such as a solver) is needed to "reproduce" the certificate besides Coq and its libraries.

We show below how a certificate is generated. Consider the cycle C_5 depicted in Figure 1. The following lines define the order of this graph and its set of vertices:

```
Definition n := 5.
Let inst_vert := 'I_n.
```

where 'I_n stands for the set $\{0, 1, \ldots, n-1\}$.
Then, the edge relationship is declared as follows:

```
Let inst_adj(u v : ℕ) :=
   match u, v with
   | 0, 1 => true
   | 0, 4 => true
   | 1, 2 => true
   | 2, 3 => true
   | 3, 4 => true
   | _, _ => false
   end.
Let inst_rel := [rel u v : inst_vert | give_sg inst_adj u v].
```

The graph is constructed from proofs of the fact that `inst_rel` is symmetric and irreflexive:

```
Let inst_sym : symmetric inst_rel. Proof. exact: give_sg_sym. Qed.
Let inst_irrefl : irreflexive inst_rel. Proof. exact: give_sg_irrefl. Qed.
Definition inst : SGraph inst_sym inst_irrefl.
```

where the definition of `give_sg` and the proofs of the properties `give_sg_sym` and `give_sg_irrefl` were performed by us, and stored in the file `prelim.v`. The instance definition is complete after declaring the weights:

```
Definition weight (v : inst) :=
   match v with
   | Ordinal 0 _ => 2
   | Ordinal 1 _ => 2
   | Ordinal 2 _ => 1
   | Ordinal 3 _ => 1
   | Ordinal _, _ => 1
   end.
```

that assigns the weights $2, 2, 1, 1, 1$ to vertices v_0, v_1, v_2, v_3 and v_4 respectively. Now, it is the time for providing the certificate. From now on, we only highlight the relevant lines of Coq code. Here, we propose a particular irredundant set of G; in this case $D = \{v_0, v_1\}$:

```
Definition inst_set := [set 'v0; 'v1].
```

Then, the facts that $w(D) = 4$ and D is indeed irredundant:

```
Fact inst_set_weight : weight_set weight inst_set = 4.
Proof. ··· Qed.
```

```
Fact inst_set_is_irr : @irredundant inst inst_set.
Proof. ··· Qed.
```

and the fact that $IR_w(G) \geq 4$:

```
Fact IR_w_lb : IR_w inst weight ≥ 4.
Proof.
   move: inst_set_weight ← ;
   apply: IR_max ;
   exact: inst_set_is_irr.
Qed.
```

whose proof uses the previous facts and `IR_max` (defined in the `dom.v` file of the graph library) which states that $w(D) \leq IR_w(G)$ for any irredundant set D of a graph G.

In the unweighted case, the certificate is shorter. No weights are given and, after the definition of the irredundant set, one declares `inst_set_is_irr` (as above) and a proof of the cardinality of D. For the same graph C_5 and $D = \{v_0, v_1\}$, we have:

```
Fact inst_set_card : #|inst_set| = 2.
Proof. ··· Qed.
```

Finally, $IR(G) \geq 2$:

```
Fact IR_lb : IR inst ≥ 2.
Proof.
   rewrite eq_IR_IR1 ; move: inst_set_card ;
      rewrite (@cardwset1 inst inst_set).
   move← ; apply: IR_max ; exact: inst_set_is_irr.
Qed.
```

where `eq_IR_IR1` means $IR(G) = IR_{\mathbf{1}}(G)$ and `cardwset1` means that $w = \mathbf{1}$ implies $|D| = w(D)$. Both are proved in `dom.v`.

In the folder `certs` one can find the certificates corresponding to the instances reported in Section 4.