Formalization of matching numbers with mathcomp-finmap and -classical

Kazunori Matsuda[1] Takafumi Saikawa[2] Yosuke Tsuji[2]

[1]Kitami Institute of Technology

[2]Nagoya University

Rocqshop 2025

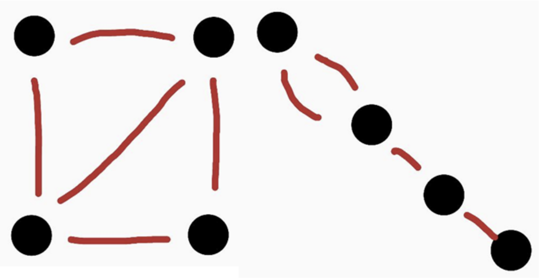
Table of contents

Motivation

Graphs

Graphs we talk about consist of vertices and edges, and are undirected:

- V : type of vertices
- E : type of edges
- d : mapping from edges to <u>sets</u> of vertices
- axiom : $|d(x)| = 2$



Motivation: relation to commutative algebra

From a graph $G = (V, E, d)$, algebraic objects can be constructed:

- a commutative polynomial ring S, by regarding vertices as variables
- an edge ideal I, by reading each edge $\{x, y\}$ as a monomial xy

It is known that some invariants of the quotient ring S/I are bounded by the graph invariants we formalized.
Towards this goal, we have managed to finish the graph theory side, leaving formalization of ideals almost untouched.

Graph invariants – Matching numbers

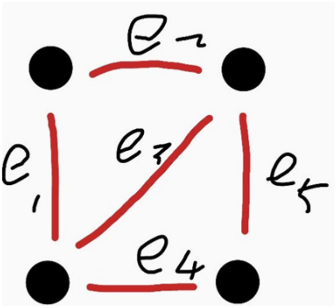
For a graph (V, E, d), a subset of edges $S \subset E$ is called a

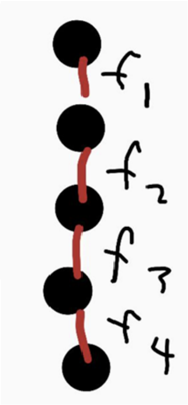
matching set	if no two edges share a vertex, and an
induced matching set	if no two edges are connected by an edge

From these sets, the following graph invariants are defined:

matching number	Maximum size of a matching set
induced matching number	Maximum size of an induced matching set
minimal matching number	<u>Minimum</u> size of a maximal matching set

Graph invariants – Matching numbers (contd.)





$\{e1, e5\}$, $\{e2, e4\}$ and singletons are matching sets for the first graph, but only the singletons are induced matching sets.
Similarly, $\{f1, f3\}$, $\{f2, f4\}$, $\{f1, f4\}$, and singletons are matching sets for the second graph, but in this case $\{f1, f4\}$ is also an induced matching.

Graph invariants – independence number

For a graph (V, E, d), a subset of vertices $S \subset V$ is called an independent set, if no two points in it do not have an edge between them.
The independence number `nindep` is the maximum size of an independent set.

In our formalization

We formalized inequalities between these invariants, which all appear not only in graph theory, but in the context of commutative algebra:

$$\texttt{nindmatch} \leq \texttt{nminmatch} \leq \texttt{nmatch} \leq 2\texttt{nminmatch}$$

$$\texttt{nindmatch} \leq \texttt{nindep}$$

$$2(\texttt{nmatch} - \texttt{nminmatch}) \leq \texttt{nindep}$$

Rocq definitions of graphs

Graph as a module

In the first attempt, we formalized graphs as a module:

```
Module LooplessUndirectedGraph.
Section def.
Record t := mk {
  V : finType;
  E : finType;
  boundary : E -> {fset V};
  _ : forall e : E, #|` boundary e | = 2;
}.
End def.
Module Exports.
Notation llugraph := t.  Notation "`d" := boundary.
Notation "`E"      := E.  Notation "`V" := V.
End Exports.
End LooplessUndirectedGraph.
Import LooplessUndirectedGraph.Exports.
```

Graph as an HB structure

We rewrote the definition using HB for future extensions:

```
HB.mixin Record isLooplessUndirectedGraph T := {
  vertex : finType;
  edge : finType;
  boundary : edge -> {fset vertex};
  size_boundary : forall e : edge, size (boundary e) = 2;
}.

#[short(type=llugraph)]
HB.structure Definition LooplessUndirectedGraph :=
  {T of isLooplessUndirectedGraph T}.

Notation "`V" := vertex.
Notation "`E" := edge.
Notation "`d" := boundary.
```

Transition to HB

To move from `Module` to `HB`, changes were needed
only in concrete examples .

First, applications of the record constructor `mk`

```
Record t := mk { V:finType; E:finType; boundary:E->{fset V};
  _ : forall e : E, #|` boundary e | = 2; }.
```

had to be replaced by `HB.instance`:

```
Definition V := 'I_2.
Definition E := 'I_2.
Definition d (_ : E) : {fset V} := fsetT.
Lemma axiom (e : E) : #|` d e | = 2.
Proof. by rewrite cardfsT card_ord. Qed.
- Definition G := LooplessUndirectedGraph.mk axiom.
+ HB.instance Definition _ :=
+   isLooplessUndirectedGraph.Build unit axiom.
+ Notation G := unit.
```

Transition to HB (contd.)

Second, we had to insert a trivial rewrite when doing a case analysis:

```
Definition V := 'I_3. (* v0, v1, v2 *)
Definition E := 'I_2. (* e0, e1 *)
Definition d (e : E) : {fset V} :=
  if e == e0 then [fset v0; v1] else [fset v1; v2].

 Example inj_boundary_is_not_necessarily_matching :
   exists S : {fset `E G}, inj_boundary S /\ ~ @is_matching G S.
 Proof.
 exists [fset: `E G]; split.
   move => e f _ _ /=.
-  rewrite /d.
+  rewrite (_ : `d = d) // /d.  (* NB: `d is a mixin field *)
  by case: ifPn; (* ... *)
```

A glitch?

```
Context {V E : finType} {d : E -> {fset V}}.
        (proof : forall e : E, #|` d e | = 2).
HB.instance Definition _ :=
  isLooplessUndirectedGraph.Build unit axiom.
Notation G := unit
```

The key type `unit` can however be anything:

```
Context (* ... *) (any_type : Type).
HB.instance Definition _ :=
  isLooplessUndirectedGraph.Build any_type axiom.
Notation G := any_type
```

I might have abused `HB`
(by not relating the mixin to its parameter)

Rocq definitions of matching sets

Finite sets and classical sets

MathComp, -Finmap and -Classical each provide a library for sets:

set in MathComp finite sets in a finite type;
list representation by enumerating `T`

fset in Finmap finite sets in a type with a choice function;
list represetantion by linear-ordering `T` by the choice

set in Classical arbitrary sets in an arbitrary type;
wrapper for `Prop`-valued predicates

Finite sets and classical sets

MathComp, -Finmap and -Classical each provide a library for sets:

set in MathComp finite sets in a finite type;
list representation by enumerating `T`

fset in Finmap finite sets in a type with a choice function;
list represetantion by linear-ordering `T` by the choice

set in Classical arbitrary sets in an arbitrary type;
wrapper for `Prop`-valued predicates

- We need the cardinality of each matching set, so it has to be finite.

Finite sets and classical sets

MathComp, -Finmap and -Classical each provide a library for sets:

set in MathComp
: finite sets in a finite type;
list representation by enumerating `T`

fset in Finmap
: finite sets in a type with a choice function;
list represetantion by linear-ordering `T` by the choice

set in Classical
: arbitrary sets in an arbitrary type;
wrapper for `Prop`-valued predicates

- We need the cardinality of each matching set, so it has to be finite.
- Finmap fsets are better than MathComp sets for compatibility with the others.

Finite sets and classical sets

MathComp, -Finmap and -Classical each provide a library for sets:

set in MathComp finite sets in a finite type;
list representation by enumerating `T`

fset in Finmap finite sets in a type with a choice function;
list represetantion by linear-ordering `T` by the choice

set in Classical arbitrary sets in an arbitrary type;
wrapper for `Prop`-valued predicates

- We need the cardinality of each matching set, so it has to be finite.
- Finmap fsets are better than MathComp sets for compatibility with the others.
- There can be a few design options for the set of all matching sets.

`matching` as Finmap and Classical sets

Here are two definitions of `matching`, the set of all matching sets.
With Finmap, we need to prepare a `bool`-valued (bracketed) predicate:

```
Definition is_matching (S : {fset `E G}) :=
  [∀ e in S, [∀ f in S, (e != f) ==> [`d e ⊥ `d f]]].
  (* [ X ⊥ Y ] = X and Y are disjoint *)
Definition matching : {fset {fset `E G}} :=
  [fset S : {fset `E G} | is_matching S].
```

`matching` as Finmap and Classical sets

Here are two definitions of `matching`, the set of all matching sets.
With Finmap, we need to prepare a `bool`-valued (bracketed) predicate:

```
Definition is_matching (S : {fset `E G}) :=
  [∀ e in S, [∀ f in S, (e != f) ==> [`d e ⊥ `d f]]]].
  (* [ X ⊥ Y ] = X and Y are disjoint *)
Definition matching : {fset {fset `E G}} :=
  [fset S : {fset `E G} | is_matching S].
```

Doing classically, the predicate can be `Prop`-valued:

```
Definition is_matching (S : {fset `E G}) :=
 ∀ e f, e ∈ S -> f ∈ S -> e != f -> [`d(e) ⊥ `d(f)].
Definition matching : set {fset `E G} :=
  [set S : {fset `E G} | is_matching S].
```

Matching numbers

The following definition of the matching number works with both Finmap or classical `matching`:

```
Definition nmatch :=  \max_(S∈matching) #|` S |.
```

Matching numbers

The following definition of the matching number works with both Finmap or classical `matching`:

```
Definition nmatch :=  \max_(S∈matching) #|` S |.
```

- The operator `\max` demands `matching` to be coercible to a finite type.

Matching numbers

The following definition of the matching number works with both Finmap or classical `matching`:

```
Definition nmatch := \max_(S ∈ matching) #|` S |.
```

- The operator `\max` demands `matching` to be coercible to a finite type.
- Finmap fset is a finite type by the coercion `fset_sub_type`

```
fset_sub_type (K : choiceType) : {fset K} -> finType
Record fset_sub_type K A : predArgType :=
  { fsval : Choice.sort K;  fsvalP : is_true (fsval \in A) }.
```

Matching numbers

The following definition of the matching number works with both Finmap or classical `matching`:

```
Definition nmatch :=  \max_(S∈matching) #|` S |.
```

- The operator `\max` demands `matching` to be coercible to a finite type.
- Finmap fset is a finite type by the coercion `fset_sub_type`

  ```
  fset_sub_type (K : choiceType) : {fset K} -> finType
  Record fset_sub_type K A : predArgType :=
    { fsval : Choice.sort K;  fsvalP : is_true (fsval \in A) }.
  ```

- Classical set is coerced to a type by `set_type`

  ```
  set_type (T : Type) : set T -> Type
  Definition set_type T A := {x : T | x \in A}
  ```

Matching numbers

The following definition of the matching number works with both Finmap or classical `matching`:

```
Definition nmatch :=  \max_(S∈matching) #|` S |.
```

- The operator `\max` demands `matching` to be coercible to a finite type.
- Finmap fset is a finite type by the coercion `fset_sub_type`
  ```
  fset_sub_type (K : choiceType) : {fset K} -> finType
  Record fset_sub_type K A : predArgType :=
    { fsval : Choice.sort K;  fsvalP : is_true (fsval \in A) }.
  ```
- Classical set is coerced to a type by `set_type`
  ```
  set_type (T : Type) : set T -> Type
  Definition set_type T A := {x : T | x \in A}
  ```
- If `T` is finite, then `set_type T A` also becomes finite by subtyping.

Other invariants

The first definition of the set of all matching sets, `matching`, is as follows:

```
Definition is_induced_matching :=
  ∀ e f, e ∈ S -> f ∈ S -> e != f ->
  [`d(e) ⊥ `d(g)] || [`d(f) ⊥ `d(g)].
Definition is_maximal_matching :=
  (S ∈ matching) /\ ∀ T : {fset `E}, (S ⊂ T) -> (T ∉ matching).
Definition induced_matching : set {fset `E G} :=
  [fset S : {fset `E G} | is_induced_matching S].
Definition maximal_matching : set {fset `E G} :=
  [fset S : {fset `E G} | is_maximal_matching S].
```

Comparing the two `matchings`

pros of Finmap

- coherence of the proof script by sticking to one library
- boolean rewriting is well-supported by MathComp

cons of Finmap

Need many reflections between `bool` and `Prop`:

```
Lemma matchingP G (S : {fset `E G})  :
  reflect
    {in S & S, forall e f, e != f -> [disjoint (`d e) & (`d f)]}
    (S \in matching G).
```

pros of Classical

No need to prepare or use the reflection; mostly just `rewrite inE`.

cons of Classical

- Insertions of `mem_set` and `set_mem` are often necessary:
  ```
  mem_set (T : Type) (A : set T) (u : T) : A u -> u \in A
  set_mem (T : Type) (A : set T) (u : T) : u \in A -> A u
  ```
- `S \notin matching` cannot be simplified by `rewrite inE`

Comparing the two `matchings` (contd.)

There was no big difference in the usability of the two.

Comparing the two `matchings` (contd.)

There was no big difference in the usability of the two.

What a boring statement.

Comparing the two `matching`s (contd.)

There was no big difference in the usability of the two.

What a boring statement.

Moreover, both have issues when formalizing subsets of `matching`:

```
Lemma induced_sub_matching G : induced_matching G `<=` matching G.
Lemma maximal_sub_matching G : maximal_matching G `<=` matching G.
```

We need to manually apply these subset lemmas, yet they look automatable.

Use HB?

Using HB to define a type of matching sets will solve the subset issues, and may provide better reasoning than using sets

```
HB.mixin Record isMatching (G : llugraph) (S : {fset `E G}) :=
  { ismatching : is_matching S }.

#[short(type=matching)]
HB.structure Definition Matching (G : llugraph) :=
{ S of isMatching G S }.

Definition nmatch := \max_(S in matching) #|` \val S |.
```

However, the last line fails because `matching` is not immediately a finite type.

Other invariants (contd.)

For a graph (V, E, d) and $S \subset V$, S is said to be an independent set if it does not contain both of the boundary vertices of any edge.

```
Definition is_independent_set :=
  [forall e : `E, ~~ (`d(e) ⊂ S)].
Definition independent_set :=
  [fset S : {fset `V} | is_independent_set S].

(* independence number;
   often denoted by α in the literature *)
Definition nindep := \max_(S∈independent_set) #|` S |.
```

Formalizing lemmas from [Hirano-Matsuda https://arxiv.org/abs/2001.10704]

```
Lemma nindmatch_leq_nindep (G : llugraph) :
  nindmatch G <= nindep G.

Lemma nmatch_minmatch_leq_nindep G :
  (nmatch G - nminmatch G).*2 <= nindep G.
```

Formalizing lemmas from [Hirano-Matsuda `https://arxiv.org/abs/2001.10704`]

```
Lemma nindmatch_leq_nindep (G : llugraph) :
  nindmatch G <= nindep G.

Lemma nmatch_minmatch_leq_nindep G :
  (nmatch G - nminmatch G).*2 <= nindep G.
```

- Pen-paper proof: 4+4 lines
- Coq proof: 6+7 lines

Concluding remarks

Why not Coq-Graph?

The Coq-Graph library by Doczkal et al. was present when we started this formalization, and we assessed if we could start using it:

- Our aim was to reason about sets of edges, and the definition of edges in terms of relations as in Coq-Graph seemed like a detour for us.
- We were about to deal only with undirected graphs, and Coq-Graph's formalization did not look direct in this respect.

A bit more of our expertize on MathComp at that time might have changed the decision to go with a new definition.

Conclusion

- MathComp, -Finmap, and -Classical together work as a practical basis for working with graph theory
- Subsets and invariants of a graph can be formalized without an extreme frustration.
- They could be however more smooth.
- TODO:
    - Connection to ring theory
    - Use Hierarchy-Builder more effectively
    - Bridge to Coq-Graph