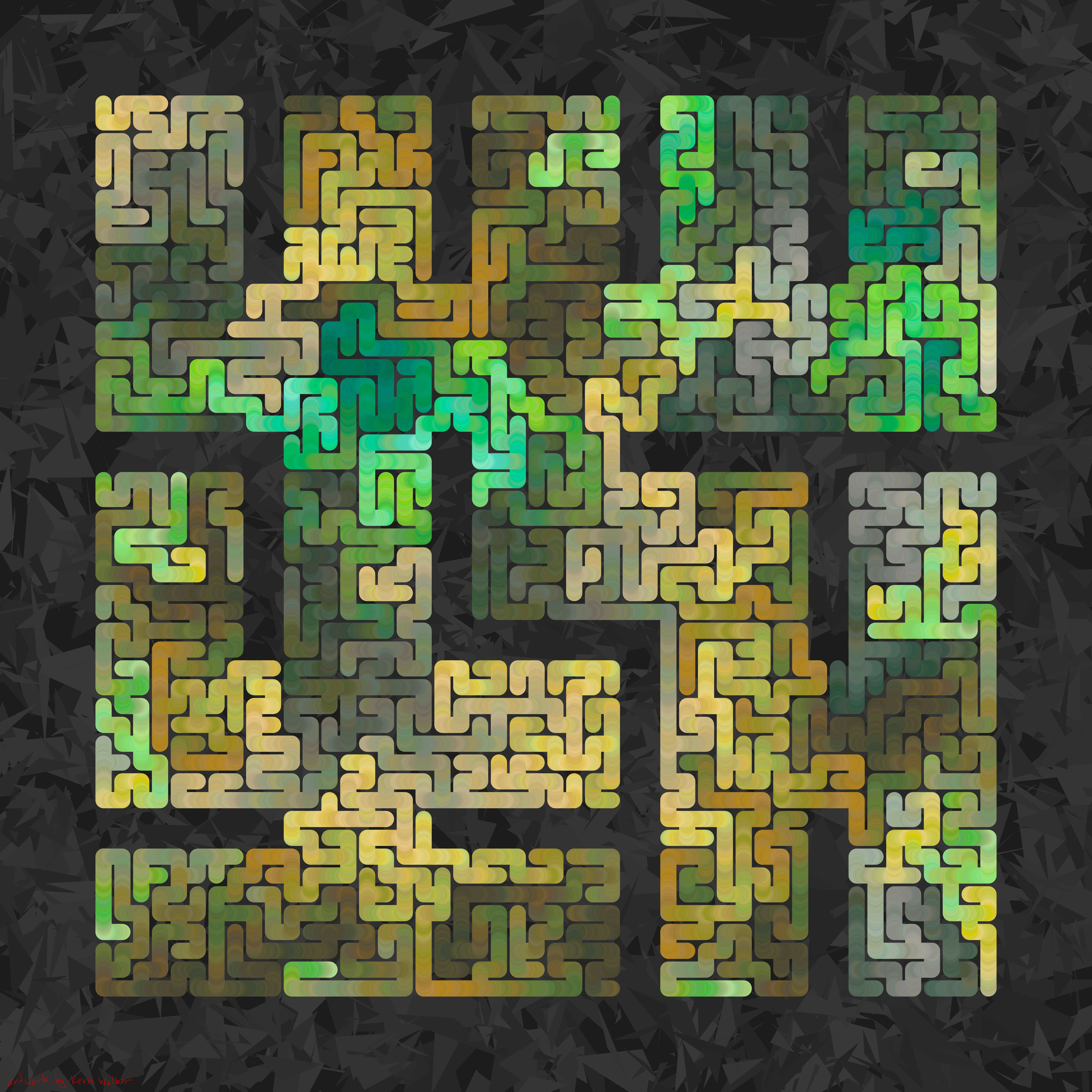

artwork by Kevin Walker

interactive theorem proving

↳ Why? To become better mathematicians.

↳ But interactive theorem proving makes it <u>harder</u> rather than <u>easier</u> to prove stuff...

— so let's work on that!

interactive theorem proving

Dreams:

- interactive style, in natural langauge
- effective automation, preserving human comprehensibility.
    - finishing tactics disposing of boring goals
    - interactive tactics that don't explode
- extensibility by users: mathematics, parsing, automation.

interactive theorem proving

Today: experiments in Lean

- dependent types, very similar to Coq
- meta-programming happens in the same language
- developed at MSR and CMU
- maths library / automation / tooling / code generation

  all 'works in progress'.
- Lean 4 out ... 'soon'?

Dependent type theory comes naturally to mathematicians:

```
structure Presheaf :=
(X : Top.{v})
(𝒪 : (open_set X) ⥤ C)

structure Presheaf_hom (F G : Presheaf.{u v} C) :=
(f : F.X ⟶ G.X)
(c : G.𝒪 ⟹ ((open_set.map f) ⋙ F.𝒪))
```

— also: "3 is not a topology on 2"

A demo?

```
theorem infinitude_of_primes (N : ℕ) : ∃ p ≥ N, prime p :=
begin
  let M := fact N + 1,
  let p := min_fac M,
  have pp : prime p, back [ne_of_gt],
  -- Adding a `#`, i.e. as `back# [ne_of_gt]`, reports the expression back built:
  -- exact min_fac_prime (ne_of_gt (succ_lt_succ (fact_pos N)))
  existsi p,
  split,
  {
    by_contradiction,
    simp at a,
    have h₁ : p ∣ M, back,
    have h₂ : p ∣ fact N, back [prime.pos, le_of_lt],
    have h : p ∣ 1, back [nat.dvd_add_iff_right],
    back [prime.not_dvd_one],
  },
  assumption,
end
```

where next?

- Lean makes it significantly easier to write new tactics
- It's still way too cumbersome to write mathematics in Lean.
- As it becomes possible for mathematicians to write tactics (not just langauge developers), this may rapidly change!
- Meta programming happens in Lean, under the [meta] keyword
- Monadic programming to interact with tactic_state
- Pattern matching and antiquotations for expr munging

where next?

Category theory in Lean

- Can we write enough automation, so that we can write 'human-like' proofs?

(i.e. omitting lots of detail!)

Unimath:

```
(** * Yoneda functor *)

(** ** On objects *)

Definition yoneda_objects_ob (C : precategory) (c : C)
          (d : C) := hom C d c.

Definition yoneda_objects_mor (C : precategory) (c : C)
    (d d' : C) (f : hom C d  d') :
   yoneda_objects_ob C c d' -> yoneda_objects_ob C c d :=
    λ g, f · g.

Definition yoneda_ob_functor_data (C : precategory) (hs: has_homsets C) (c : C) :
    functor_data (C^op) HSET.
Proof.
  exists (λ c', hSetpair (yoneda_objects_ob C c c') (hs c' c)) .
  intros a b f g. unfold yoneda_objects_ob in *. simpl in *.
  exact (f · g).
Defined.


Lemma is_functor_yoneda_functor_data (C : precategory) (hs: has_homsets C) (c : C) :
  is_functor (yoneda_ob_functor_data C hs c).
Proof.
  repeat split; unf; simpl.
  unfold functor_idax .
  intros.
  apply funextsec.
  intro f. unf. apply id_left.
  intros a b d f g.
  apply funextsec. intro h.
  apply (! assoc _ _ _ ).
Qed.

Definition yoneda_objects (C : precategory) (hs: has_homsets C) (c : C) :
             functor C^op HSET :=
    tpair _ _ (is_functor_yoneda_functor_data C hs c).


(** ** On morphisms *)

Definition yoneda_morphisms_data (C : precategory)(hs: has_homsets C) (c c' : C)
    (f : hom C c c') : ∏ a : ob C^op,
         hom   (yoneda_objects C hs c a) ( yoneda_objects C hs c' a) :=
```

Coq:

```
Section yoneda_lemma.
  Context `{Funext}.
  Variable A : PreCategory.
  Variable G : object (A^op -> set_cat).
  Variable a : A.
    (** There is a contravariant version of Yoneda's lemma which
        concerns contravariant functors from [A] to [Set]. This
        version involves the contravariant hom-functor

        [hₐ = Hom(—, A)],

        which sends [x] to the hom-set [Hom(x, a)]. Given an arbitrary
        contravariant functor [G] from [A] to [Set], Yoneda's lemma
        asserts that

        [Nat(hₐ, G) ≅ G(a)]. *)

  Definition yoneda_lemma_morphism
  : morphism set_cat
             (BuildhSet
                (morphism (A^op -> set_cat) (yoneda A a) G)
                _)
             (G a)
    := fun phi => phi a 1%morphism.

  Local Arguments Overture.compose / .

  Definition yoneda_lemma_morphism_inverse
  : morphism set_cat
             (G a)
             (BuildhSet
                (morphism (A^op -> set_cat) (yoneda A a) G)
                _).
  Proof.
    intro Ga.
    hnf.
    let F0 := match goal with |- NaturalTransformation ?F ?G => constr:(F) end in
    let G0 := match goal with |- NaturalTransformation ?F ?G => constr:(G) end in
    refine (Build_NaturalTransformation
              F0 G0
              (fun a' : A => (fun f : morphism A a' a => morphism_of G f Ga))
              _
           ).
    simpl in *.
    abstract (
```

Isabelle:

```
theory Yoneda
imports NatTrans SetCat
begin

definition YFtorNT' C f ≡ ⦇NTDom = Hom_C[−,dom_C f] , NTCod = Hom_C[−,cod_C f] ,
                NatTransMap = λ B . Hom_C[B,f]⦈

definition YFtorNT C f ≡ MakeNT (YFtorNT' C f)

lemmas YFtorNT-defs = YFtorNT'-def YFtorNT-def MakeNT-def

lemma YFtorNTCatDom: NTCatDom (YFtorNT C f) = Op C
by (simp add: YFtorNT-defs NTCatDom-def HomFtorContraDom)

lemma YFtorNTCatCod: NTCatCod (YFtorNT C f) = SET
by (simp add: YFtorNT-defs NTCatCod-def HomFtorContraCod)

lemma YFtorNTApp1: assumes X ∈ Obj (NTCatDom (YFtorNT C f)) shows
(YFtorNT C f) $$ X = Hom_C[X,f]
proof−
 have (YFtorNT C f) $$ X = (YFtorNT' C f) $$ X using assms by (simp add:
MakeNTApp YFtorNT-def)
 thus ?thesis by (simp add: YFtorNT'-def)
qed

definition
  YFtor' C ≡ ⦇
        CatDom = C ,
        CatCod = CatExp (Op C) SET ,
        MapM = λ f . YFtorNT C f
  ⦈

definition YFtor C ≡ MakeFtor(YFtor' C)

lemmas YFtor-defs = YFtor'-def YFtor-def MakeFtor-def

lemma YFtorNTNatTrans':
  assumes LSCategory C and f ∈ Mor C
  shows NatTransP (YFtorNT' C f)
proof(auto simp only: NatTransP-def)
  have Fd: Ftor (NTDom (YFtorNT' C f)) : (Op C) ⟶ SET using assms
    by (simp add:  HomFtorContraFtor Category.Cdom YFtorNT'-def)
  have Fc: Ftor (NTCod (YFtorNT' C f)) : (Op C) ⟶ SET using assms
    by (simp add:  HomFtorContraFtor Category.Ccod YFtorNT'-def)
  show Functor (NTDom (YFtorNT' C f)) using Fd by auto
  show Functor (NTCod (YFtorNT' C f)) using Fc by auto
  show NTCatDom (YFtorNT' C f) = CatDom (NTCod (YFtorNT' C f))
    by(simp add: YFtorNT'-def NTCatDom-def HomFtorContraDom)
  show NTCatCod (YFtorNT' C f) = CatCod (NTDom (YFtorNT' C f))

        MapM = λ f . YFtorNT C f
  ⦈
```

```
variables (C : Type u₁) [𝒞 : category.{u₁ v₁} C]
include 𝒞

def yoneda : C ⥤ ((Cᵒᵖ) ⥤ (Type v₁)) := λ X, λ Y : C, Y ⟶ X.

def yoneda_evaluation : (((Cᵒᵖ) ⥤ (Type v₁)) × (Cᵒᵖ)) ⥤ (Type (max u₁ v₁)) :=
(evaluation (Cᵒᵖ) (Type v₁)) ⋙ ulift_functor.{v₁ u₁}

@[simp] lemma yoneda_evaluation_map_down
  (P Q : (Cᵒᵖ ⥤ Type v₁) ×  (Cᵒᵖ)) (α : P ⟶ Q) (x : (yoneda_evaluation C) P) :
  ((yoneda_evaluation C).map α x).down = (α.1) (Q.2) ((P.1).map (α.2) (x.down)) := rfl

def yoneda_pairing : (((Cᵒᵖ) ⥤ (Type v₁)) × (Cᵒᵖ)) ⥤ (Type (max u₁ v₁)) :=
let F := (category_theory.prod.swap ((Cᵒᵖ) ⥤ (Type v₁)) (Cᵒᵖ)) in
let G := (functor.prod ((yoneda C).op) (functor.id ((Cᵒᵖ) ⥤ (Type v₁)))) in
let H := (functor.hom ((Cᵒᵖ) ⥤ (Type v₁))) in
  (F ⋙ G ⋙ H)

@[simp] lemma yoneda_pairing_map
  (P Q : (Cᵒᵖ ⥤ Type v₁) ×  (Cᵒᵖ)) (α : P ⟶ Q) (β : (yoneda_pairing C) (P.1, P.2)) :
  (yoneda_pairing C).map α β = (yoneda C).map (α.snd) ≫ β ≫ α.fst := rfl

def yoneda_lemma : (yoneda_pairing C) ≅ (yoneda_evaluation C) :=
{ hom := { app := λ F x, ulift.up ((x.app F.2) (𝟙 F.2)) },
  inv := { app := λ F x, { app := λ X a, (F.1.map a) x.down } } }.
```

How does this work?

- an approximation of Ganesalingam-Gowers 'human-style automation'
  in Lean (arXiv:1309.4501)

- an algorithm for automatic rewriting,
  using an edit distance heuristic and
  some machine learning.

  (in progress, w/ Keeley Hoek, ANU)

rewrite_search proves equational goals by rewriting subexpressions using specified (or discovered) lemmas

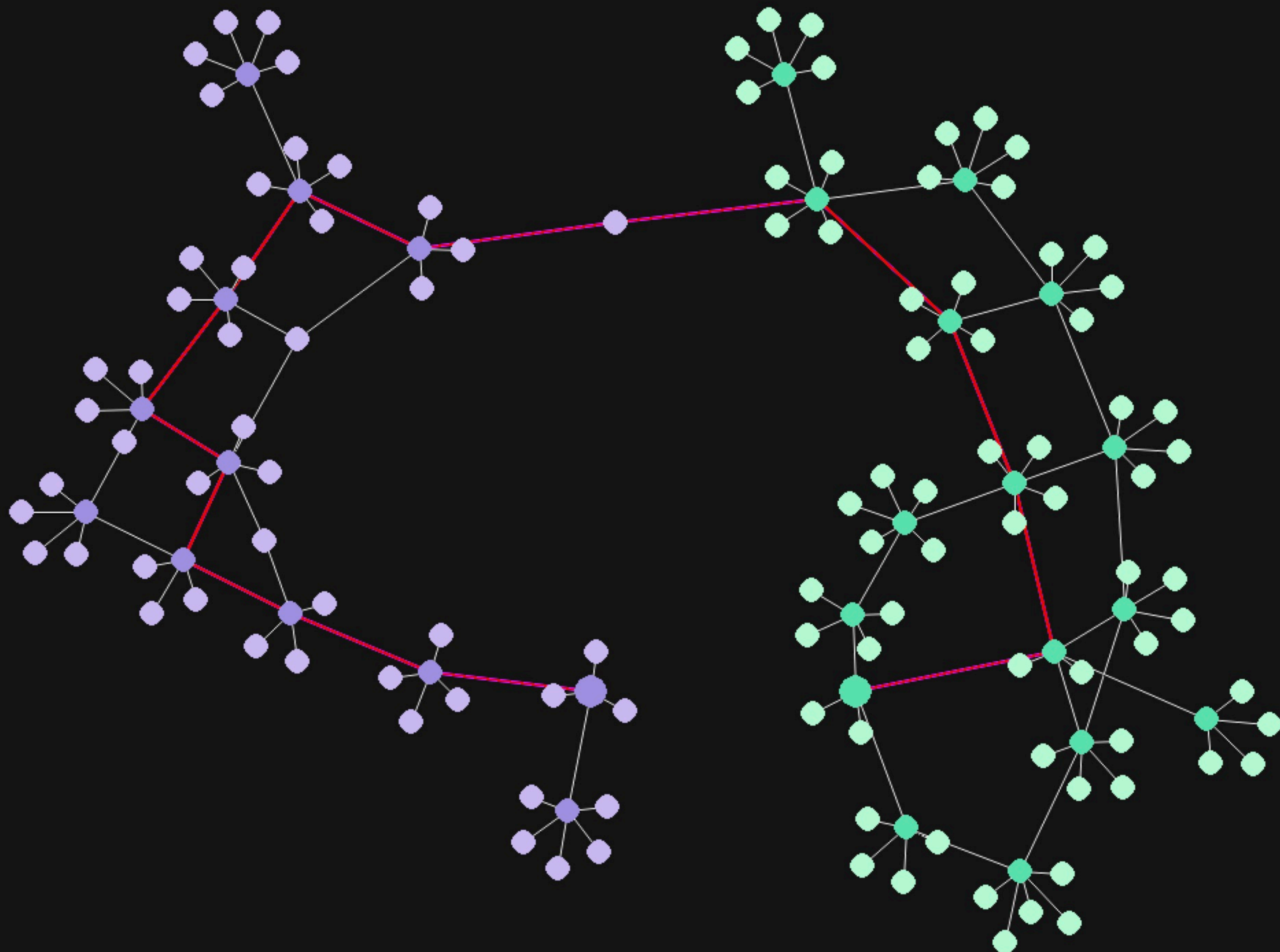
A depth or breadth first search of the rewrite graph would be hopeless for all but the most trivial goals.

The basic version of rewrite_search uses an edit distance minimising search

- We search from both sides of the goal $A = B$ simultaneously.
- Pretty print each side, and calculate edit distances.
- We track a list of 'interesting pairs', A', B' with small edit distance
- At each step we consider a rewrite of A' or B', for the most interesting pair at that point.
- In the basic version, most interesting means smallest edit distance $d(A', B')$.

Generalisations

- use A^* rather than greedy search
  (so most interesting is the minimiser of
  $$d(A,A') + d(A',B') + d(B',B).)$$
- modify edit distance
  - look at tokens appearing in A and B, and run a classifier on them.
  - increase the edit distance weighting for significant tokens
  - dynamically update weights during the search, based on tokens in all $\{A_i\}$ and $\{B_i\}$.
  - centre-of-mass classifier, or use libsvm in a modified version of Lean.



100 : 161/25/22

5: 82/19/15

3: 65/18/13

svm: 70/18/13

2: 62/24/19