In this third project our objective is to introduce the box product of two simple graphs, and prove that $G \square H$ is connected if and only if G and H are connected.

1 Graph connectivity

Lean's mathlib already includes simple graphs and some basic constructions such as walks and paths. However, it does not have a definition for connectivity and some corresponding important results which we first need to introduce.

A graph G with vertex set V_G is set to be connected if $V_G \neq \emptyset$ and $\forall u, v \in V_G$ there exists a path from u to v. In Lean this is written as:

```
def is_connected {V : Type} (G : simple_graph V) : Prop :=
  nonempty V ∧ ∀ u v : V, ∃ p : G.walk u v, p.is_path
```

The next thing is that graph isomorphisms preserve walks and thus connectivity. This is rather obvious as a graph isomorphism $\phi : G \to H$ preserves the vertex and edge set. To show this in Lean we recursively decompose the walk in G to reconstruct it in H using the mapping of the adjacency relation of the vertices from G to H:

```
/-- Graph isomorphisms preserve walks -/
def graph_iso.walk {V W : Type} {G : simple_graph V} {H : simple_graph W}
  (e : G ≃g H) : Π {a b : V}, G.walk a b → H.walk (e(a)) (e(b))
| _ _ simple_graph.walk.nil := walk.nil
| _ _ (simple_graph.walk.cons hGadj Gwalk) :=
  walk.cons (e.map_adj_iff.mpr hGadj) (graph_iso.walk Gwalk)
```

The proof for connectivity follows from this, as if we have a walk one can always produce a path. This is done using the mathlib `simple_graph.walk.to_path` function. The other part of connectivity is to show that the vertex set of H is non empty. Which is obvious as if there exists a vertex $v \in G$ then $\phi(v) \in H$

Currently, the proof for connectivity relies on the `classical` tactic as the function `simple_graph.walk.to_path` relies on the vertex set of H to be an instance of `decidable_eq`. One improvement might be to remove this dependency by creating a lemma that states graph isomorphism preserve paths without needing to rely on the `simple_graph.walk.to_path` function.

2 The box product

The box product, $G\square H$, of two graphs G and H is defined as follows:

- $G\square H$ has vertex set $V_{G\square H} = V_G \times V_H$
- There is an edge between (g_1, h_1) and (g_2, h_2) iff
  - Either $h_1 = h_2$ and there is an edge between g_1 and g_2 in G,
  - Or $g_1 = g_2$ and there is an edge between h_1 and h_2 in H.

Which in Lean is defined as a `simple_graph` structure:

```
def box_product {V W : Type} (G : simple_graph V) (H : simple_graph W) :
  simple_graph (V × W) :=
{
  -- a b : V × W;
  -- `.1` corresponds to an element of G `.2` corresponds to an element of H
  adj := λ (a b), (a.2 = b.2 ∧ G.adj a.1 b.1) ∨ (a.1 = b.1 ∧ H.adj a.2 b.2),
  symm := -- ...
  loopless := -- ...
}
```

To be able to use the notation $G\square H$ in Lean we need to define a new infix operator:

```
infix ` □ `:70 := box_product
```

We use 70 as the precedence so that it has a higher precedence compared to graph (iso)morphisms. This doesn't allow us to remove the brackets when we use the field dot notation, and we don't know if there is a way to fix this.

We proceed to write a basic API to be able to rewrite the adjacency relation of the box product whose proofs are `refl`. Following up by proving that the box product is commutative and associative up to isomorphism. The proofs are rather straightforward albeit tedious. They are defined as followed:

```
/-- The box product is commutative up to isomorphism -/
def box_product_comm {V W : Type} (G : simple_graph V) (H : simple_graph W) :
  G□H ≃g H□G := -- ...

/-- The box product is associative up to isomorphism -/
def box_product_assoc {U V W : Type} (G : simple_graph U) (H : simple_graph V)
  (K : simple_graph W) : (G□H)□K ≃g G□(H□K) := -- ...
```

We then create an API to move adjacency relations of G from itself to the graph $G \square H$:

```
/- Adjacency relations to move between the simple graph and the box product -/
lemma adj_lhs_equiv {V W : Type} {a b : V} {y : W} {G : simple_graph V}
  {H : simple_graph W} : G.adj a b ↔ (G□H).adj (a, y) (b, y) :=
begin
  split, {
    -- Lift from G to G□H
    intro hGadj,
    left,
    rw [eq_self_iff_true, true_and],
    exact hGadj,
  }, {
    -- Projection from G□H to G
    intro hGHadj,
    cases hGHadj with hGB hHB, {
      exact hGB.2,
    }, {
      -- H is a simple graph so there is no edge between w and w
      -- This is the condition irrefel,
      -- and_false simplifies the hyp as if one side of an AND
      -- is false than the prop is false
      simp only [irrefl, and_false] at hHB,
      exfalso,
      exact hHB,
    }
  }
end
```

The same is done for adjacency relations between H and $G \square H$.

The lifting of a walk from G to $G \square H$ is straightforward as one can just trace the walk in $(g_1, \ldots, g_n)$ for a constant vertex $h \in H$, by using the adjacency relation between G and $G \square H$.

```
def lift_walk_lhs {V W : Type} {G : simple_graph V} {H : simple_graph W}
  (y : W) : Π {a b : V}, (G.walk a b) → (G□H).walk (a, y) (b, y)
| _ _ simple_graph.walk.nil := walk.nil
| a b (simple_graph.walk.cons hGadj Gwalk)
  := walk.cons (adj_lhs_equiv.mp hGadj) (lift_walk_lhs Gwalk)
```

Lifting a walk from H is done similarly.

The projection/descent of a walk from $G \square H$ to G is more complicated. The basic idea is that if we are moving along an edge in H we discard this step of the walk and only keep the components which move along edges in G.

In Lean we managed to do the part which kept the edges moving along in G in tactic mode, but could not figure out how to discard the edges along H. Thanks to Kenny Lau showing us `or.by_cases`, which is how one can do the `cases` tactic in term mode, and `show ... by rw ...` which allows to do some rewrite in term mode, we could finalise and convert to term mode the projection of a walk. The use of `or.by_cases` requires the types and adjacency relations to be decidable, therefore requiring us to add some instance of `[decidable_eq T]` and `[decidable_rel G.adj]`.

Put all together we get the following Lean code:

```
def descend_walk_lhs {V W : Type} [decidable_eq V] [decidable_eq W]
  {G : simple_graph V} [decidable_rel G.adj]
  {H : simple_graph W} [decidable_rel H.adj]
  : Π {vw1 vw2 : V × W}, (G□H).walk vw1 vw2 → (G.walk vw1.1 vw2.1)
| _ _ simple_graph.walk.nil := walk.nil
| vw1 vw3 (simple_graph.walk.cons hGHadj p) :=
  or.by_cases hGHadj (λ hBG, walk.cons hBG.2 (descend_walk_lhs p))
  (λ hBH, show G.walk vw1.1 vw3.1, by rw hBH.1; exact descend_walk_lhs p)
```

The projection of a walk from $G\square H$ to H is done similarly.

3 $G\square H$ is connected if and only if G and H are connected

We are now ready to show that $G\square H$ is connected $\iff$ G and H are connected.

3.1 $G\square H$ is connected if G and H are connected

The proof is straightforward:

Proof 1
Let $(g,h) \in V_{G\square H} = V_G \times V_H$. By the connectedness of H, $\forall k,l \exists$ path from (g_i,h_k) to (g_i,h_l). Similarly, by the connectedness of G, $\forall i,j \exists$ path from (g_i,h_k) to (g_j,h_k).

Therefore, there exists a walk (and thus a path) from (g_i,h_k) to (g_j,h_l) by concatenating the path from (g_i,h_k) to (g_j,h_k) and the path from (g_j,h_k) to (g_j,h_l).

The Lean proof follows this structure, with the additional easy proof that the vertex set of the box product $V_G \times V_H$ is not empty.

3.2 $G\square H$ is connected only if G and H are connected

The tricky aspect of the proof for this direction has already been solved with the projection of the walks from $G\square H$ to G (respectively H).