# On Trees and Their Topological Realisations in Homotopy Type Theory - Draft

Jonathan Prieto-Cubides
Department of Informatics
University of Bergen
Bergen, Norway
jonathan.cubides@uib.no

## Abstract

In this work, we characterise in homotopy type theory the type of rooted trees and oriented spanning trees, which are the generalisation of the notions of a tree and spanning tree for directed multigraphs. We state and prove the theorem about the mere existence of oriented spanning trees for connected graphs with a finite node set and a set of edges between any two nodes. In addition, we look at the topological realisation for graphs, which in HoTT can be seen as a coequalizer, the higher-inductive type that considers a topological space where the nodes are points and edges are paths glued to their endpoints. In this view, a graph is connected if its geometric realisation is a connected type. A tree is a graph with no non-trivial loops, for which its topological realisation is connected and contractible. Finally, a proof is given to show that the realisation of rooted trees is a connected and contractible type, as expected. We believe the results here can help to study the fundamental group of a graph, which requires computing a spanning tree. A formalisation accompanies most results in Agda using the Cubical mode and the Cubical Agda library.

***Keywords:*** spanning trees, topological realisation, univalent mathematics

**ACM Reference Format:**
Jonathan Prieto-Cubides. 2022. On Trees and Their Topological Realisations in Homotopy Type Theory - Draft. *Proc. ACM Program. Lang.* 1, X, Article 1 (January 2022), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

This paper is part of an ongoing effort to formalise some concepts and results from Graph theory in Univalent mathematics. Here, in particular, we introduce the concept of spanning trees, conditions for their mere existence and their connection with the topological realisation of a graph in Homotopy Type Theory. A spanning tree of a connected undirected graph is a subgraph that is a tree and spans the graph; that is, it includes all the graph's nodes [1]. Not every graph has a spanning tree, at least in a constructive setting. Only finite connected graphs have. Nevertheless, constructing spanning trees plays a relevant role as an intermediate step in computing free groups from graphs and several other algorithms like pathfinding and network protocols. On the former, we find Swan's proof of the Nielsen–Schreier theorem in HoTT. The development of Sections 4 and 4.1 is inspired by Lemma 4.16 [9, §4]. However, from the topological point of view, Swan defines trees and spanning trees in terms of the geometric realisation of a graph using coequalizers. Then, one could say our journey ends when Swan's work starts. Consequently, we focus on the combinatorial aspects of graphs, constructing trees in a more familiar way to graph theorists. Finally, we connect our constructions with the topological view in Section 5.

Throughout the rest of the paper, we assume the reader is familiar with our chosen mathematical foundation, homotopy type theory. We follow a derived notation from the HoTT Book [11] and one previous work [6, §2]. Additionally, a few excerpts of the computer formalisation in Agda [10] accompany most definitions and lemmas; however, many details are omitted. For the complete proofs, see the Agda formalisation, available on the following website.

https://jonaprieto.github.io/synthetic-graph-theory/cubical.

***Outline.*** This paper is structured as follows. In Section 2, we briefly comment on the chosen formalisation tool, Agda and its Cubical mode. Section 3 provides a few basic definitions used throughout the rest of the paper. In Section 4, we formally state lemmas to enlarge subtrees and prove the mere existence of spanning trees for certain graphs. In Section 5, we prove that the topological realisation of a rooted tree, as introduced in Section 3, is connected and contractible. Finally, the paper concludes with some remarks and future work in Section 6.

## 2 Computer Formalisation

The formalisation of the mathematical content of this work is carried out in Cubical Agda [12], a language extension for Agda to support Cubical type theory. In contrast to Homotopy type theory, Cubical Agda and other cubical type

Author's address: Jonathan Prieto-Cubides, Department of Informatics, University of Bergen, Postboks 7803, Thormøhlens Gate 55, Bergen, 5020, Norway, jonathan.cubides@uib.no.

2022. 2475-1421/2022/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

theories give computational meaning to Voevodsky's Univalence and other aspects like higher-inductive types. This characteristic makes it possible to prove principles like function and propositional extensionality easily.

It is worth mentioning that our development could be carried out in vanilla Agda with HoTT support. However, Cubical Agda provides a more convenient way to formalise a few proofs, as the results in Section 5. A distinguishing feature of Cubical Agda is its native support for a big family of higher-inductive types (HITs), which adds judgmental computation rules for all constructors. In Cubical Agda, it is possible to define (dependent) functions on HITs by matching all patterns, i.e., including point and path constructor patterns. However, the same definitions for HITs and their functions are genuinely rather tedious in Agda without the Cubical Mode since it requires, for example, one to state a set of postulates for its computational rules and elimination principles; see Licata's trick and the HoTT-Agda library. Alternatively, in recent versions of Agda, it is possible to extend the capabilities of the Agda type-checker by adding custom rewriting rules. This feature can be used to alleviate the lack of support for HITs in Agda without the Cubical Mode.

This document is an example of literate programming. The Agda CLI can generate high-quality LaTeX files based on a mixture of textual content and Agda code. To start using Cubical Agda, one needs to use the flag `--cubical`. The other flag, `--guardedness`, is only necessary to type check the coinduction definition given at the end of Section 4.1. We have also included imports to use the Cubical Agda library [5] and one local module with a few small lemmas needed later.

```
{-# OPTIONS --cubical --guardedness #-}
open import Cubical.Core.Everything
open import Base
```

## 3 Basic Concepts

Let us start defining a few basic concepts needed for the rest of this development.

### 3.1 The Type of Graphs

**Definition 3.1** (`Graph`)**.** A graph consists of a type of nodes equipped with a binary type valued relation of edges.

$$\mathsf{Graph} :\equiv \Sigma_{(N:\mathcal{U})}(N \to N \to \mathcal{U}). \tag{3.1}$$

Here we define the type of graphs using a record type in Agda for convenience.

```
record Graph : Type (ℓ-suc ℓ) where
  constructor graph
  field
    N : Type ℓ
    E : N → N → Type ℓ
open Graph
```

### 3.2 The Type of Walks

On the other hand, the type of walks in a graph can be defined as an indexed inductive data type, similarly to the polymorphic type for lists. Such an inductive type is sometimes convenient in formalising results on walks [6] since it allows us to define walk functions by pattern matching easily. Unfortunately, pattern matching is not supported in Cubical Agda for such inductive data types at the moment of writing. We, therefore, consider the following equivalent types from where the former type is chosen for the convenience of the lemmas stated in this document. In particular, walks here grow by attaching edges at their ends, as in Lemma 4.3. In what follows, we denote by $W_G^n(x, y)$ the type of walks from $x$ to $y$ of length $n$ in a graph $G$.

1. Walks formed by backwards edge addition.

```
W : ℕ → N G → N G → Type ℓ
W 0 x y = x ≡ y
W (suc n) a c = Σ[ b ∈ N G ] (W n a b) × (E G b c)
```

2. Walks formed by forward edge addition.

```
W' : ℕ → N G → N G → Type ℓ
W' 0 x y = x ≡ y
W' (suc n) a c = Σ[ b ∈ N G ] (E G a b) × (W' n b c)
```

As typical in HoTT, once a type is defined, one would like to characterise its identity type. One can prove that the identity type for graphs coincides with the type of isomorphisms. In the case of walks, we compute the identity type point-wise. However, since we are only interested in the case where graphs consist of sets, the type of walks of such graphs turns out to be a set, which follows from Lemma 3.2.

**Lemma 3.2** (`W-is-set`)**.** *Let $G$ be a graph such that the type of nodes is a set and the family of edges consists of sets. Then, the type of walks of length $n$ from $x$ to $y$ is a set, for any $x, y : N_G$ and $n : \mathbb{N}$.*

A proof term for this lemma in Agda is the following.

```
module _ (V-is-set : isSet (N G))
         (E-is-set : (x y : N G) → isSet (E G x y)) where
  W-is-set : (n : ℕ) → (x y : N G) → isSet (W n x y)
  W-is-set zero _ _ = isProp→isSet (V-is-set _ _)
  W-is-set (suc n) _ _ = isOfHLevelΣ 2 V-is-set λ _ →
    (isOfHLevel× 2 (W-is-set n _ _ ) (E-is-set _ _))
```

We often work with strongly connected graphs throughout the following lemmas unless otherwise stated. Let us define such a property as the mere existence of a walk between any pair of nodes.

**Definition 3.3** (`isGConnected`)**.** A graph $G$ is strongly connected if the type in (3.2) is inhabited.

$$\mathsf{isGConnected}(G) :\equiv \Pi_{(x,y:N_G)} \|\Sigma_{(n:\mathbb{N})} W_G^n(x, y)\|. \tag{3.2}$$

In Agda, the type above is defined as follows.

```
isGConnected : Graph → Type ℓ
isGConnected G = (x y : N G) → ∥ Σ[ n ∈ ℕ ] W G n x y ∥
```

**Lemma 3.4.** *Being connected for a graph is a proposition.*

### 3.3 Rooted Trees and Subgraphs

Trees are usually defined as undirected graphs with a single path between any pair of nodes. However, we prefer to use a more suitable notion of a tree for working directly with directed multigraphs. Therefore, we consider the class of rooted trees, which are directed graphs with a single node acting as the root of the tree and a single walk between any pair of nodes.

The notion of trees for directed graphs can also be defined in terms of zig-zags, which are walks formed by edges of different possible orientations. In this view, a tree is then a graph if the corresponding type of zig-zag walks is contractible. Finally, it is worth mentioning that the definition of the type of undirected graphs and other derived concepts, including trees and trails, can be found in Agda–UniMath [8]. In this Agda library, an undirected graph consists of a type $V$ of nodes and a family $E$ of types over the unordered pairs of $V$. Lastly, an unordered pair of elements in a type $A$ consists of a two-element type $X$ and a map of type $X \to A$.

Let us now define the type of rooted trees in a directed multigraph $G$. We refer to rooted trees as trees in the rest of this work unless otherwise stated.

**Definition 3.5** (`isTree`)**.** A graph $G$ is a *tree* if the type in (3.3) is contractible. The node in the centre of contraction is referred to as the *root* of the tree.

$$\Sigma_{(r:\mathsf{N}_G)} \Pi_{(x:\mathsf{N}_G)} \mathsf{isContr}(\Sigma_{(n:\mathbb{N})} \mathsf{W}^n_G(r, x)) \tag{3.3}$$

In Agda, the type of rooted trees is defined as follows.

```
isTree : Graph → Type ℓ
isTree G = isContr(Σ[ r ∈ N G ] (∀ x → isContr(Σ[ n ∈ ℕ ] W G n r x)))
```

**Lemma 3.6** (`isProp-isTree`)**.** *Being a tree is a proposition.*

We need now to define the notions of subgraph and subtree. Recall that we are interested in defining and constructing spanning trees out of strongly connected graphs, which are trees containing all nodes of the original graph. If the graph is finite and strongly connected, such trees can be obtained by traversing the graph using a depth-first search or a breadth-first search (BFS)algorithm. For a more general class of graphs, a principle of choice may be needed to guide the search. In Section 4.1, we prove that a spanning tree merely exists if the node set of the graph is a type inhabited and the graph is strongly connected with a family of discrete sets as the type of edges.

**Definition 3.7** (`Subgraph`)**.** A subgraph of $G$ is a graph $H$ with an embedding into $G$, denoted by $H \hookrightarrow G$. The type of subgraphs of $G$ is $\mathsf{Subgraph}(G)$.

$$\begin{aligned}\mathsf{Subgraph}(H, G) :\equiv \Sigma_{((h,g):\mathsf{Hom}(H,G))} \mathsf{isEmbedding}(h) \\ \times \Pi_{(x,y:\mathsf{N}_H)}, \mathsf{isEmbedding}(g(x, y)),\end{aligned}$$

where $\mathsf{Hom}(H, G)$ is the type of graph homomorphisms from $H$ to $G$ and isEmbedding is the property that the function ap/`cong` is an equivalence, as defined in the HoTT Book.

Almost faithfully, we define in Agda the above structure on graphs as follows.

```
module _ {ℓ : Level} (G : Graph {ℓ}) where
  record Subgraph (H : Graph {ℓ}) : Type ℓ where
    field
      h : N H → N G
      g : (x y : N H) → E H x y → E G (h x) (h y)
      h-is-emb : isEmbedding h
      g-is-emb : (x y : N H) → isEmbedding (g x y)
```

**Definition 3.8** (`isSubtree`)**.** A (decidable) subtree of $G$ is a tree and subgraph of $G$ equipped with a mechanism for checking whether a node in $G$ is in it or not.

```
record isSubtree (H : Graph {ℓ}) : Type ℓ where
  constructor subtree
  field
    is-subgraph : Subgraph H
    is-tree : isTree H
    dec-fiber : (x : N G) → Dec (fiber (Subgraph.h is-subgraph) x)
```

## 4 Enlarging Subtrees

Let us develop a few lemmas about the notion of a subgraph and subtree about how to construct larger subtrees out of subgraphs. The main result of this section is Lemma 4.3, which requires first to state the following crucial lemma.

**Lemma 4.1** (`∃-edgecut`)**.** *Let $G$ be a connected graph such that its node set is partitioned into two disjoint nonempty types $V_1$ and $V_2$. Then, it merely exists an edge connecting a node of $V_1$ to some node of $V_2$ and vice versa.*

*Proof.* Since we want to prove a proposition, let us apply the elimination principle of the propositional truncation to the fact of $G$ being connected. One can obtain a function $f$, which returns a walk connecting any two nodes of $G$. Let $v_1, v_2$ be nodes in $V_1$ and $V_2$, respectively, and $w$ be the walk obtained by $f(v_1, v_2)$.

Let us proceed by induction on the length of $w$. We will exhibit an edge in the walk $w$ that must have one node in $V_1$ and the other node in $V_2$, as illustrated in Figure 1. If the walk has zero length, then there is nothing to prove since such a case is impossible by construction. Then, we can assume the induction hypothesis holds for a walk of length $n$.

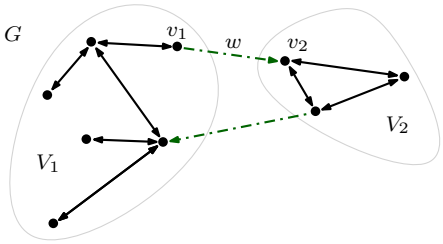


**Figure 1.** The walk $w$ in Lemma 4.1's proof.

Let $p \cdot e$ be a walk of length $n+1$ where $p$ is a walk from $x$ to $y$ and $e$ is an edge from $y$ to $v_2$. Since the node set of $G$ is equivalent to $V_1 + V_2$, the node $y$ is either in $V_1$ or $V_2$. If $y$ is in $V_1$, the required edge is $e$. Otherwise, we get the required edge by induction on the walk $p$. □

**Figure 2.** The term `∃-edgecut` defined below is the Agda term for the Lemma 4.1's proof.

```
module EdgeCutLemma {ℓ : Level} {V₁ V₂ : Type ℓ}
  (G : Graph {ℓ}) (G-is-connected : isGConnected G)
  (e : N G ≃ V₁ + V₂)
  (v₁ : V₁) (v₂ : V₂) where


  ∃-edgecut : ∥ Σ[ x ∈ V₁ ] Σ[ y ∈ V₂ ] E G (from-V₁ x) (from-V₂ y) ∥
  ∃-edgecut = trunc-elim isPropPropTrunc (λ {(n , w) → f v₁ v₂ n w}) w
    where
    isoN : Iso (N G) (V₁ + V₂)
    isoN = equivToIso e

    w : ∥ Σ[ n ∈ N ] W G n (from-V₁ v₁) (from-V₂ v₂) ∥
    w = G-is-connected _ _

    f : (a : V₁) (b : V₂) (n : N) → W G n (from-V₁ a) (from-V₂ b)
      → ∥ Σ[ x ∈ V₁ ] Σ[ y ∈ V₂ ] E G (from-V₁ x) (from-V₂ y) ∥
    f _ _ zero w = ⊥-elim (inlinr-→-⊥ (isoInvInjective isoN _ _ w))
    f v₁ v₂ (suc n) (b , w , ed)
      with from-NG b | inspect from-NG b
    ... | inl x      | [ from-NGb≡inlx ]
      = | x , v₂ , subst (λ o → E G o _) helper ed |
        where
        helper : b ≡ from-V₁ x
        helper = sym (retEq e b) ∙ cong to-NG from-NGb≡inlx
    ... | inr x | [ from-NGb≡inrx ]
      = f v₁ x n (subst (λ o → W G n _ o) helper w)
        where
        helper : b ≡ from-V₂ x
        helper = sym (retEq e b) ∙ cong to-NG from-NGb≡inrx
```

**Lemma 4.2** (`decompose-image`). *Let $A, B : \mathcal{U}$ and $f$ be an embedding from $A$ to $B$ such that the type of fibers $\mathsf{fib}_f(x)$ is a decidable set for any $x : B$. Then, the following equivalence holds.*

$$B \simeq A + \Sigma_{(x:B)} \neg\, \mathsf{fib}_f(x),$$

*where* $\mathsf{fib}_f(b) :\equiv \Sigma_{(a:A)} f(a) = b$.

**Lemma 4.3** (`∃-subtree`). *Let $G$ be a connected graph with a discrete node set such that each type of edges $\mathsf{E}_G(x,y)$ is a set for any pair of nodes $x, y$. If $H$ is a subtree of $G$ such that there is a node $u$ in $H$ and a node $v$ in $G$ but not in $H$, then there merely exists a subtree of $G$ enlarging $H$ with one additional node.*

*Proof.* Since $H$ is a subtree, then, there must be a pair $(h, g) : H \hookrightarrow G$. We can decompose the set of nodes of $G$ as in (4.1) by applying Lemma 4.2 to the embedding $h$ and the fact that the set of nodes of $H$ is a discrete set. We write $\mathsf{N}_{G\setminus H}$ for the set $\Sigma_{(x:\mathsf{N}_H)} \mathsf{fib}_h(x)$.

$$\mathsf{N}_G \simeq \mathsf{N}_H + \mathsf{N}_{G\setminus H}. \tag{4.1}$$

Let $p$ be of type $\|\Sigma_{(x:\mathsf{N}_H)} \Sigma_{(y:\mathsf{N}_{G\setminus H})} \mathsf{E}_G(x,y)\|$, obtained by applying Lemma 4.1 to the fact that $G$ is connected, and the node set of $G$ is partitioned as the coproduct of two nonempty sets. The sets $\mathsf{N}_H$ and $\mathsf{N}_{G\setminus H}$ are nonempty by assumption. Now, since the goal of this proof is a proposition, by eliminating of the propositional truncation applied to $p$, we can assume that there is an edge $e$ from a node in $H$ to some node in $\mathsf{N}_{G\setminus H}$. Finally, by Lemma 4.12, the graph $H$ can be extended by adding to it the edge $e$ to get the subgraph $H^*$ of $G$, similarly as illustrated in Figure 4. The definition of $H^*$ is given in Definition 4.4. The proof $H^*$ is a subtree of $G$ is given in Lemma 4.12. □

The remainder of this section is devoted to supporting the construction of the extended subtree $H^*$ of $G$, which is crucial for the proof of Lemma 4.3. The definition of $H^*$ is given in Definition 4.4. The proofs that $H^*$ is a subgraph and a subtree are given in Lemmas 4.6 and 4.12, respectively. We assume below that $H$ is a subgraph of $G$, defined by $(h, g) : H \hookrightarrow G$. Additionally, there is a designated edge $\hat{e}$ from $\hat{x}$ in $H$ to $\hat{y}$ in $G$. The node $\hat{y}$ is not in $H$, as illustrated in Figure 4. As a matter of notation, the singleton graph formed by the node $x$ with no edges is denoted by $\{x\}$.

**Definition 4.4.** The graph obtained from adding to $H$ the edge $\hat{e}$ is referred as to $H^*$. Formally speaking, the set of nodes $\mathsf{N}_{H^*}$ is the set $\mathsf{N}_H + \{\hat{y}\}$ and the family of edges in $H^*$ is defined below. Recall that the function $h$, appearing below in (4.2), is the embedding from $\mathsf{N}_H$ to $\mathsf{N}_{H^*}$ given by the fact that $H$ is a subgraph of $G$.

$$\mathsf{E}_{H^*}(x,y) :\equiv \begin{cases} \mathsf{E}_H(a,b) & \text{if } x \equiv \mathsf{inl}(a), y \equiv \mathsf{inl}(b), \\ h(a) = h(\hat{x}) & \text{if } x \equiv \mathsf{inl}(a), y \equiv \mathsf{inr}(b), \\ \mathbb{0} & \text{otherwise.} \end{cases} \tag{4.2}$$

**Figure 3.** An excerpt of the Agda term for Lemma 4.3.

```
module _ (G : Graph {ℓ})
         (G-is-connected : isGConnected G)
         (_≟Node_ : (x y : N G) → Dec (x ≡ y))
         (E-is-set : (x y : N G) → isSet (E G x y)) where
  ∃-subtree
    : (H : Graph)
    → (H-subtree : isSubtree G H)
    → (u : N H) → (v : N G)
    → ¬ (fiber (Subgraph.h (isSubtree.is-subgraph H-subtree)) v)
    → ‖ Σ[ H* ∈ Graph ] isSubtree G H* × (N H* ≃ (N H + 𝟙)) ‖
  ∃-subtree H H-subtree u v v-not-in-H =
    trunc-elim isPropPropTrunc helper ∃-edgecut
    where
    H-subgraph = isSubtree.is-subgraph H-subtree
    h = Subgraph.h H-subgraph
    h-is-emb = Subgraph.h-is-emb H-subgraph
    h-has-dec-image = isSubtree.dec-fiber H-subtree
    V₁ = N H


    isoN : N G ≃ V₁ + V₂
    isoN = decompose-image _ _ h h-is-emb h-has-dec-image

    open EdgeCutLemma G G-is-connected
                      isoN u (v , v-not-in-H) hiding (E*)

    helper : Σ[ x ∈ V₁ ] Σ[ y ∈ V₂ ] E G (from-V₁ x) (from-V₂ y) → _
    helper (x , y , ed) = | H* , H*-subtree , e' |
      where
      -- H* is the graph obtained by adding an edge to H.
      -- H*-subtree is a term constructed in Lemma 4.5-4.16
```

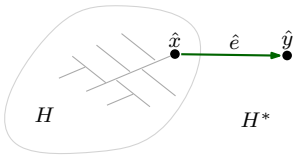


**Figure 4.** The graph $H^*$, mentioned in Lemmas 4.5 to 4.12, obtained by adding an edge $\hat{e}$ to $H$. The edge $\hat{e}$ is given by Lemma 4.1.

**Lemma 4.5.** *Let $H^*$ be the graph defined in Definition 4.4. The following properties hold for $a, b : \mathsf{N}_H$ and $c : \mathsf{N}_{\{\hat{y}\}}$.*

1. *The type $\mathsf{E}_{H^*}(\mathsf{inl}(a), \mathsf{inr}(b))$ is a proposition.*
2. *The type $\mathsf{E}_{H^*}(\mathsf{inl}(\hat{x}), \mathsf{inr}(c))$ is contractible.*
3. *The type $\Sigma_{(a:\mathsf{N}_H)} \mathsf{E}_{H^*}(\mathsf{inl}(a), \mathsf{inr}(\hat{y}))$ is contractible.*

**Lemma 4.6** (H*-subgraph)**.** *The graph $H^*$ is a subgraph of $G$.*

*Proof.* To show that $H^*$ is a subgraph of $G$, it suffices to provide an embedding $h^* : \mathsf{N}_{H^*} \to \mathsf{N}_G$ and a function $g^* : \Pi_{(x,y:N_H)} \mathsf{E}_{H^*}(x, y) \to \mathsf{E}_G(h(x), h(y))$ such that for all $x, y : N_H$, the function $g^*(x, y)$ is an embedding.

Since $H$ is a subgraph of $G$, let $(h, g) : H \hookrightarrow G$, as stated in Definition 3.7.

$$h^*(x) :\equiv \begin{cases} h(a) & \text{if } x :\equiv \mathsf{inl}(a) \text{ for } a : \mathsf{N}_H, \\ \hat{y} & \text{otherwise.} \end{cases}$$

It is clear that $h^*$ is an embedding, since when restricting to $H$, it is the embedding $h$. Otherwise, it is a map from a contractible domain, which is clearly an embedding.

Finally, let $g^* : \Pi_{(a,b:\mathsf{N}_{H^*})} \mathsf{E}_{H^*}(a, b) \to \mathsf{E}_{H^*}(h^*(a), h^*(b))$ be the mapping on edges in $H^*$ defined as follows.

$$g^*(x, y, e) :\equiv \begin{cases} g(a, b, e) & \text{if } x \equiv \mathsf{inl}(a), y \equiv \mathsf{inl}(b), \\ \mathsf{tr}(\mathsf{ap}_h(h^{-1}(e)), \hat{e}) & \text{if } x \equiv \mathsf{inl}(a), y \equiv \mathsf{inr}(b), \\ & \text{and } e : h(a) = h(\hat{x}), \\ \mathbb{0} & \text{otherwise.} \end{cases}$$

By definition, the function $g^*$ restricted to $H$ is the embedding $g$. Otherwise, the next corresponding nontrivial case is $g^*(\mathsf{inl}(a), \mathsf{inr}(b))$. By Lemma 4.5-(1), it is possible to show that any fiber of $g^*(\mathsf{inl}(a), \mathsf{inr}(b))$ is a proposition, and it is then an embedding. In any case, we conclude that $g^*(x, y)$ is an embedding, from where the conclusion follows. □

To prove Lemmas 4.11 and 4.12, we need to show a few intermediate results, which we now state. In Lemmas 4.7 to 4.9, let $n : \mathbb{N}$ and $a, b$ be two nodes in $H$.

**Lemma 4.7.** *The following equivalence holds.*

$$W^n_H(a, b) \simeq W_{H^*}(\mathsf{inl}(a), \mathsf{inl}(b)). \tag{4.3}$$

**Lemma 4.8.** *The following types are empty.*

1. $W^n_{H^*}(\hat{y}, \mathsf{inl}(a))$.
2. $\Pi_{(v:\mathsf{N}_H)} \mathsf{isContr}(W^n_{H^*}(\mathsf{inl}(a), \mathsf{inl}(v)))$.
3. $\Sigma_{(n:\mathbb{N})} W^{n+1}_{H^*}(\hat{y}, \hat{y})$.

**Lemma 4.9.** *The following types are contractible.*

1. $W^0_{H^*}(\hat{y}, \hat{y})$.
2. $\Sigma_{(n:\mathbb{N})} W^n_{H^*}(\hat{y}, \hat{y})$.

**Lemma 4.10.** *The type in (4.4) is empty.*

$$\Sigma_{(y:\{\hat{y}\})} \Pi_{(v:\mathsf{N}_{H^*})} \mathsf{isContr}\left(\Sigma_{(n:\mathbb{N})} W^n_{H^*}(\mathsf{inr}(y), v)\right). \tag{4.4}$$

*Proof.* It suffices to show that there is no walk from $y$ to some node in $H$. Let $y$ be a node in $\{\hat{y}\}$ and $v$ be a node in $H^*$.

$$P(y, v) :\equiv \mathsf{isContr}(\Sigma_{(n:\mathbb{N})} W^n_{H^*}(\mathsf{inr}(y), v)).$$

Then,

$$\begin{aligned} &\Sigma_{(y:\{\hat{y}\})} \Pi_{(v:\mathsf{N}_{H^*})} P(y, v) \\ &\simeq \Pi_{(v:\mathsf{N}_{H^*})} P(\hat{y}, v) \\ &\simeq \Pi_{(v:\mathsf{N}_H)} P(\hat{y}, \mathsf{inl}(v)) \times \Pi_{(v:\{\hat{y}\})} P(\hat{y}, \mathsf{inr}(v)) \\ &\simeq \mathbb{0} \times \Pi_{(v:\{\hat{y}\})} P(\hat{y}, \mathsf{inr}(v)) \\ &\simeq \mathbb{0}. \end{aligned}$$

□

**Lemma 4.11** (Bottleneck). *Let $G$ be a connected graph, $H$ be a subtree of $G$ with root $\mathbf{r}_H$. Then, there is a unique walk in the graph $H^*$ from $\mathsf{inl}(\mathbf{r}_H)$ to $\hat{y}$.*

*Proof.* It suffices to show that the following type is contractible.

$$\Sigma_{(n:\mathbb{N})} W^n_{H^*}(\mathsf{inl}(\mathbf{r}_H), \hat{y}). \tag{4.5}$$

Then,

$$\begin{aligned}
&\Sigma_{(n:\mathbb{N})} W^n_{H^*}(\mathsf{inl}(\mathbf{r}_H), \hat{y}) \\
&\simeq W^0_{H^*}(\mathbf{r}_H, \hat{y}) + \Sigma_{(n:\mathbb{N})} W^{n+1}_{H^*}(\mathbf{r}_H, \hat{y}) \\
&\simeq \mathbb{0} + \Sigma_{(n:\mathbb{N})} W^{n+1}_{H^*}(\mathbf{r}_H, \hat{y}) \\
&\simeq \Sigma_{(n:\mathbb{N})} \Sigma_{(v:\mathsf{N}_{H^*})} W^n_{H^*}(\mathsf{inl}(\mathbf{r}_H), v) \times \mathsf{E}_{H^*}(v, \hat{y}) \\
&\simeq \Sigma_{(n:\mathbb{N})} \Big( \big(\Sigma_{(v:\mathsf{N}_H)} W^n_{H^*}(\mathsf{inl}(\mathbf{r}_H), \mathsf{inl}(v)) \times \mathsf{E}_{H^*}(\mathsf{inl}(v), \hat{y})\big) \\
&\qquad + \big(\Sigma_{(v:\{\hat{y}\})} W^n_{H^*}(\mathsf{inl}(\mathbf{r}_H), \mathsf{inr}(v)) \times \mathsf{E}_{H^*}(\mathsf{inr}(v), \hat{y})\big) \Big) \\
&\simeq \Sigma_{(n:\mathbb{N})} \big( (\Sigma_{(v:\mathsf{N}_H)} W^n_H(\mathbf{r}_H, v) \times \mathsf{E}_{H^*}(\mathsf{inl}(v), \hat{y}) \\
&\qquad + \big(W^n_{H^*}(\mathsf{inl}(\mathbf{r}_H), \hat{y}) \times \mathbb{0}\big) \big) \\
&\simeq \Sigma_{(v:\mathsf{N}_H)} \big(\Sigma_{(n:\mathbb{N})} W^n_H(\mathbf{r}_H, v)\big) \times \mathsf{E}_{H^*}(\mathsf{inl}(v), \hat{y}) \\
&\simeq \Sigma_{(v:\mathsf{N}_H)} \mathbb{1} \times \mathsf{E}_{H^*}(\mathsf{inl}(v), \hat{y}) \\
&\simeq \Sigma_{(v:\mathsf{N}_H)} \mathsf{E}_{H^*}(\mathsf{inl}(v), \hat{y}) \\
&\simeq \mathbb{1}.
\end{aligned}$$

□

**Lemma 4.12** (`H*-subtree`). *The graph $H^*$ is a subtree of $G$.*

*Proof.* To show that $H^*$ is a subtree, the following must hold:

1. The graph $H^*$ is a connected subgraph of $G$, i.e., there is an embedding from $H^*$ to $G$ given as a pair of mappings $(h^*, g^*)$, as in Definition 3.7.
2. The type of fibers $\mathsf{fib}_{h^*}(x)$ is a decidable set for any node $x$ in $G$.
3. The following type is contractible.

$$\Sigma_{(r:\mathsf{N}_{H^*})} \Pi_{(v:\mathsf{N}_{H^*})} \mathsf{isContr}\left(\Sigma_{(n:\mathbb{N})} W^n_{H^*}(u, v)\right). \tag{4.7}$$

The first condition is satisfied by Lemma 4.6. Since $H$ is a subgraph of $G$, we have access to the embedding given by $(h, g) : H \hookrightarrow G$. Then, the second condition follows, since the type in question is equivalent to the $\mathsf{fib}_h(b) + (\hat{y} = b)$ for any $b$ in $G$, by the following calculation, and any equivalence of types preserve any property.

$$\begin{aligned}
\mathsf{fib}_{h^*}(b) &:\equiv \Sigma_{(a:\mathsf{N}_{H^*})} h^*(a) = b \\
&\simeq \big(\Sigma_{(a:\mathsf{N}_H)} h^*(\mathsf{inl}(a)) = b\big) + \Sigma_{(a:\{\hat{y}\})} h^*(\mathsf{inr}(a)) = b \\
&\simeq \big(\Sigma_{(a:\mathsf{N}_H)} h(a) = b\big) + (\hat{y} = b) \\
&\simeq \mathsf{fib}_h(b) + (\hat{y} = b).
\end{aligned}$$

The mapping $h^*$ has a decidable image inherited from $h$, since $H$ is a tree, and the nodes of $H$ form a discrete set. Finally, for the third condition, we have the following calculation. For brevity, let $P$ be a shorthand for the type family in (4.7).

$$\begin{aligned}
&\Sigma_{(r:\mathsf{N}_{H^*})} \Pi_{(v:\mathsf{N}_{H^*})} P(H^*, r, v) \\
&\simeq \Sigma_{(r:\mathsf{N}_H)} \Pi_{(v:\mathsf{N}_{H^*})} P(H^*, \mathsf{inl}(r), v) \\
&\qquad + \Sigma_{(r:\{\hat{y}\})} \Pi_{(v:\mathsf{N}_{H^*})} P(H^*, \mathsf{inr}(r), v) \\
&\simeq \Sigma_{(r:\mathsf{N}_H)} \Pi_{(v:\mathsf{N}_{H^*})} P(H^*, \mathsf{inl}(r), v) + \mathbb{0} \\
&\simeq \Sigma_{(r:\mathsf{N}_H)} \Pi_{(v:\mathsf{N}_{H^*})} P(H^*, \mathsf{inl}(r), v) \\
&\simeq \Sigma_{(r:\mathsf{N}_H)} \big(\Pi_{(v:\mathsf{N}_H)} P(H^*, \mathsf{inl}(r), \mathsf{inl}(v)) \\
&\qquad \times \Pi_{(v:\{\hat{y}\})} P(H^*, \mathsf{inl}(r), \mathsf{inr}(v))\big) \\
&\simeq \Sigma_{(r:\mathsf{N}_H)} \left(\Pi_{(v:\mathsf{N}_H)} P(H, r, v) \times P(H^*, \mathsf{inl}(r), \hat{y})\right) \\
&\simeq \Sigma_{\left((r,!):\Sigma_{(a:\mathsf{N}_H)} \Pi_{(v:\mathsf{N}_H)} P(H,a,v)\right)} P(H^*, \mathsf{inl}(r), \hat{y}) \\
&\simeq P(H^*, \mathsf{inl}(\mathbf{r}_H), \hat{y}) \\
&\equiv \mathsf{isContr}\left(\Sigma_{(n:\mathbb{N})} W^n_{H^*}(\mathsf{inl}(\mathbf{r}_H), \hat{y})\right) \\
&\simeq \mathsf{isContr}(\mathbb{1}) \\
&\simeq \mathbb{1}.
\end{aligned}$$

□

## 4.1 Oriented Spanning Trees

In graph theory, any connected undirected graph has at least one spanning tree. In our setting, we can prove that any strongly connected and directed multigraph has at least one oriented spanning tree.

**Definition 4.13** (`isSpannigTree`). An oriented spanning tree of $G$ is a subtree that contains all the vertices of $G$.

```
record isSpannigTree (H : Graph) : Type ℓ where
  open isSubtree; open Graph
  field
    is-subtree : isSubtree G H
  h = Subgraph.h (is-subgraph is-subtree)
  g = Subgraph.g (is-subgraph is-subtree)
  field
    cover-all-nodes : isEquiv h
```

We are ready now to prove the main result of this section.

**Lemma 4.14.** *Let $G$ be a nonempty strongly connected graph such that the node set of $G$ is finite and the family of edges of $G$ consists of sets. Then there merely exists an oriented spanning tree of $G$.*

*Proof.* Let $n$ be the cardinality of the node set of $G$. We proceed by induction on $n$. If $n = 1$, then the graph has only one node, and its spanning tree is the same one-point graph with no edges. Otherwise, let $n > 1$. We state the induction hypothesis as the mere existence of a subtree of $G$ with $k$ nodes where $k < n$. Since the goal of the lemma is a proposition, we can apply the elimination principle of the truncation to the induction hypothesis to get a subtree of $G$ with $n - 1$ nodes, namely, $H_{n-1}$. Finally, since there is a missing node of $G$ not in $H_{n-1}$, we can apply Lemma 4.3 to $G$ and $H_{n-1}$ to obtain the required spanning tree, a graph $H_n$ including all the nodes of $G$. □

The previous proof suggests that Lemma 4.14 can be generalized to the case where the node set of $G$ has a principle of choice. One can construct a chain of subtrees, ordered

by the subgraph relation, using a construction similar to the argument in Lemma 4.14's proof. Then, the spanning tree of the infinite graph is the maximal element in such a chain, assuming the axiom of choice, see Lemma 4.7 [9, §4]. However, we do not attempt to formalize this generalization here.

On the other hand, one version of the Kőnig's lemma states that if an infinite graph is locally finite and connected, then the graph contains a *ray*. A ray is a simple walk that starts at one node and continues from it through infinitely many nodes. It seems natural to consider a proof of this result using Lemma 4.1 and the axiom of choice. This direction is, however, left for future work. Here we only give a first proposal for the type of rays. A ray in the current setting can be defined as an infinite walk starting at the node $x$ such that the type of occurrences of $x$ in the walk is contractible. We can define these definitions in Agda as follows.

```
record InfiniteWalk (x : N G) : Type ℓ where
  coinductive
  field
    head : Σ[ y ∈ N G ] E G x y
    tail : InfiniteWalk (fst head)
open InfiniteWalk

{-# TERMINATING #-}
_∈w_ : (x : N G) → {y : N G} → (w : InfiniteWalk y) → Type ℓ
_∈w_ x {y} w = (x ≡ y) + (x ∈w tail w)

isRay : (x : N G) → InfiniteWalk x → Type ℓ
isRay x w = isContr (x ∈w w)
```

## 5 Topological Realisation of Graphs

The one-cell topological realisation of a graph can be represented by the coequalizer of the corresponding source and target functions. Every node in the graph is mapped to a point in the space. Moreover, any edge in the graph gives rise to a path in the space glued to the endpoints.

This topological point of view for representing graphs is further described in type theory by Swan [9]. It is worth noting that the type of graphs in this paper is equivalent to the type of graphs in their setting, as the following equivalence shows.

$$\begin{aligned}\mathsf{Graph} :\equiv \Sigma_{(N:\mathcal{U})}(N \to N \to \mathcal{U}) \\ \simeq \Sigma_{(N:\mathcal{U})}(N \times N \to \mathcal{U}) \\ \simeq \Sigma_{(N,E:\mathcal{U})}(E \to (N \times N)) \\ \simeq \Sigma_{(N,E:\mathcal{U})}((E \to N) \times (E \to N)).\end{aligned}$$

Therefore, one benefit of working in Univalent mathematics is that one can transport their results to the setting of this paper and vice versa. Now, back to Cubical Agda, let us define the topological realisation of a graph $G$ as the following higher inductive data type.

```
module realisation {ℓ : Level} (G : Graph {ℓ}) where
  data 𝕋¹ : Type ℓ where
    𝕟 : N G → 𝕋¹
    𝕖 : ∀ {a b} → E G a b → 𝕟 a ≡ 𝕟 b
```

To prove a few properties of this geometric realisation below, we define two handy elimination principles into propositions.

```
elimProp
  : {B : 𝕋¹ → Type ℓ}
  → ((x : 𝕋¹) → isProp (B x))
  → ((a : N G) → B (𝕟 a))
  → (x : 𝕋¹) → B x
elimProp _ f (𝕟 a) = f a
elimProp B-fiber-prop f (𝕖 {a}{b} e i) =
  isOfHLevel→isOfHLevelDep 1 B-fiber-prop (f a) (f b) (𝕖 e) i
```

For the particular case of relations, we obtain the following elimination principle.

```
elimPropRel
  : {R : 𝕋¹ → 𝕋¹ → Type ℓ}
  → ((x y : 𝕋¹) → isProp (R x y))
  → ((a b : N G) → R (𝕟 a) (𝕟 b))
  → (x y : 𝕋¹) → R x y
elimPropRel Rprop f =
  elimProp (λ x → isPropΠ (λ y → Rprop x y))
           (λ x → elimProp (λ y → Rprop (𝕟 x) y) (f x))
```

The walks in the graph give rise to paths in the geometric realisation, as shown in the following Agda code. As a consequence, the connectedness of a graph implies the connectedness of its geometric realisation.

```
𝕨 : {n : ℕ} {a b : N G} → W G n a b → 𝕟 a ≡ 𝕟 b
𝕨 {zero} a=b = cong 𝕟 a=b
𝕨 {suc _} (_ , w , e) = (𝕨 w) ∙ 𝕖 e
```

The realisation of walks using the function $\mathbb{w}$ respects the concatenation of walks. In particular, it respects backward edge addition, as in the Agda code below.

```
comp-edge
  : {a b c : N G} {n : ℕ}
  → (w : W G n a b) (e : E G b c)
  → 𝕨 ((_ , w , e)) ≡ (𝕨 w) ∙ (𝕖 e)
comp-edge {n = zero} w e = reflc
comp-edge {n = suc n} (_ , w , e₁) e₂ =
  cong (λ x → x ∙ (𝕖 e₂) ) (comp-edge w e₁)
```

Let us introduce the following notions to not clash with the names of some definitions defined earlier.

**Definition 5.1.** A graph is topologically connected if its geometric realisation is connected.

```
isConnected : Type ℓ → Type ℓ
isConnected A = (x y : A) → ∥ x ≡ y ∥

isTConnected : Graph → Type ℓ
isTConnected G = isConnected (realisation.𝕋¹ G)
```

**Lemma 5.2.** *Being connected for the realisation of a graph is a proposition.*

```
isProp-isTConnected : (G : Graph) → isProp (isTConnected G)
isProp-isTConnected _ = isPropΠ λ _ → isPropΠ λ _ → isPropPropTrunc
```

**Lemma 5.3.** *Being connected for a graph implies its geometric realisation is connected.*

```
isGConnected-isTConnected
  : (G : Graph) → isGConnected G → isTConnected G
isGConnected-isTConnected G G-is-connected =
  elimPropRel (λ _ _ → isPropPropTrunc) helper
  where
    open realisation G
    helper : (a b : N G) → ∥ 𝕟 a ≡c 𝕟 b ∥
    helper a b = trunc-elim isPropPropTrunc
                            (λ {(_ , w) -> ∣ 𝕨 w ∣})
                            (G-is-connected a b)
```

**Definition 5.4.** A graph is a topological tree if its geometric realisation is contractible.

```
isTopTree : Graph → Type ℓ
isTopTree G = isContr (realisation.𝕋¹ G)
```

Using this topological point of view for graphs, we can prove that any tree, as in Definition 3.5 is topologically connected and tree in a topological way. The converse is not true; see, for example, the triangle graph, where an edge connects any pair of nodes. The realisation of such a graph contains a non-trivial loop and thus is not contractible.

**Lemma 5.5.** *If the graph is tree then it is topologically connected.*

*Proof.* For this proof, we are only interested in what happens when we apply the geometric realisation on nodes and how the nodes are glued. Since the graph is a tree, we have access to its root node equipped with a walk to every other node, see Definition 3.5. Finally, one can use the walks given by the tree to connect the nodes in the geometric realisation, as illustrated in Figure 5 and proved in Agda code below. □

```
module _ {ℓ : Level}(G : Graph {ℓ}) where
  open realisation
  open walk-concat G
```

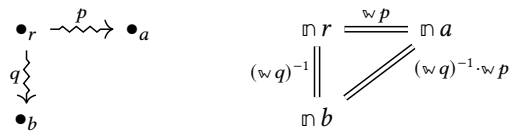


**Figure 5.** It is shown the walks and paths mentioned in Lemma 5.5's proof. The node $r$ on the left represents the root of the given tree. The node $a$ is the node connected to $r$ by the walk $p$, and similarly, the node $b$ is the node connected to $r$ by the walk $q$. Then, we can connect the realisation of $a$ and $b$ by the walk $\mathbb{w}(q)^{-1} \cdot \mathbb{w}(p)$.

```
  isTree-isTConnected : isTree G → isConnected (𝕋¹ G)
  isTree-isTConnected ((r , unique-walk-from-r-to) , _) =
    elimPropRel G ((λ _ _ → isPropPropTrunc)) helper
    where
      helper : (a b : N G) → ∥ 𝕟 {G = G} a ≡c 𝕟 b ∥
      helper a b = ∣ (sym (𝕨 G (snd p))) • 𝕨 G (snd q) ∣
        where
          p : Σ[ n ∈ N ] W G n r a
          p = fst (unique-walk-from-r-to a)
          q : Σ[ n ∈ N ] W G n r b
          q = fst (unique-walk-from-r-to b)
```

**Lemma 5.6.** *If the graph is a tree and the topological realisation is a set, then the graph is a topological tree.*

```
isTree-isSet-isTopTree : isTree G → isSet (𝕋¹ G) → isTopTree G
isTree-isSet-isTopTree
  G-is-graph-tree@((r , unique-walk-from-r-to) , _)
  𝕋¹G-is-set = 𝕟 r , λ y →
    trunc-elim (𝕋¹G-is-set (𝕟 r) y)
               (λ nr=y → nr=y)
               (isTree-isTConnected G-is-graph-tree (𝕟 r) y)
```

Finally, we can prove that a tree, in a combinatorial way, is topologically a tree.

**Lemma 5.7.** *Being a tree for a graph implies its realisation is a contractible type.*

*Proof.* Let $G$ be a graph tree. Then, we must show that $\mathbb{T}(G)$ is a contractible type. To show that, let $\mathbb{n}(r)$ be the centre of contraction of $\mathbb{T}(G)$, where $r$ is the root of $G$. Then, we must construct a function that returns a path from $\mathbb{n}(r)$ to $a$ for any $a : \mathbb{T}(G)$. We do this by induction on the constructors of $\mathbb{T}(G)$. The first case is the point constructor $\mathbb{n}(x)$ for $x : \mathsf{N}_G$, for which we can just return the realisation of the unique walk from $r$ to $x$ given by the proof that $G$ is a tree. The second and last case is the path constructor case. Given a path $\mathbb{e}(e)$, where $e$ is an edge from $a$ to $b$ in $G$, we must construct a path from $\mathbb{n}(r)$ to every point in the path $\mathbb{e}(e)$. Since $G$ is a tree, we have access to a unique walk from the

root $r$ to the nodes $a$ and $b$, respectively. Let $p$ and $q$ be such walks, as illustrated in Figure 6. Then, the required path can be obtained considering the path $\mathbb{w}(p) \cdot \mathbb{e}(e)$.

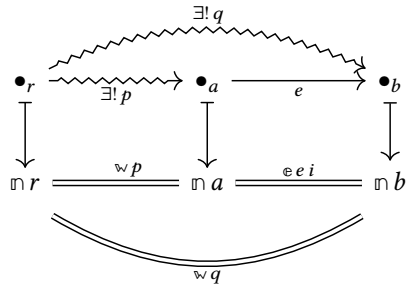


**Figure 6.** The construction of a path from $\mathbb{n}(r)$ to any point in the path $\mathbb{e}(e)$.

However, for coherence, we must make sure that there is a homotopy between the paths $\mathbb{w}(p) \cdot \mathbb{e}(e)$ and $\mathbb{w}(q)$, which is the right face of the cube as illustrated in Figure 7. The back face is the whole square of deforming the path $\mathbb{w}(p)$ to $\mathbb{w}(p) \cdot \mathbb{w}(q)$, which is precisely Lemma `compPath-filler` in the Cubical Agda library. □

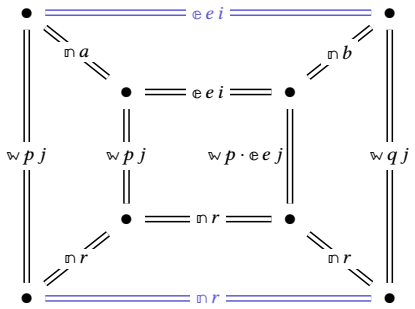


**Figure 7.** The constructed cube for Lemma 5.7's proof.

**Figure 8.** An Agda term for Lemma 5.7.

```
isTree-isTopTree : isTree G → isTopTree G
isTree-isTopTree ((r , unique-walk-from-r-to) , _) =
  𝕟 r , helper
  where
    walk = snd
    helper : (x : 𝕋¹ G) → 𝕟 r ≡c x
    helper (𝕟 x) = 𝕨 G (walk (fst (unique-walk-from-r-to x)))
    helper (𝕖 {a}{b} e i) j
      = hcomp (λ k → λ { (i = i0) → 𝕨p j
                       ; (i = i1) → 𝕨p·𝕖e≡𝕨q k j
                       ; (j = i0) → reflc {x = 𝕟 r} i
                       ; (j = i1) → 𝕖 e i
                       })
        (compPath-filler 𝕨p (𝕖 e) i j)
      where
      p : Σ[ n ∈ ℕ ] W G n r a
      p = fst (unique-walk-from-r-to a)
      length-walk-p = fst p

      q : Σ[ n ∈ ℕ ] W G n r b
      q = fst (unique-walk-from-r-to b)

      𝕨p : 𝕟 r ≡ 𝕟 a
      𝕨p = 𝕨 G (walk p)

      𝕨q : 𝕟 r ≡ 𝕟 b
      𝕨q = 𝕨 G (walk q)

      q-is-unique : q ≡c (suc (length-walk-p) , _ , walk p , e)
      q-is-unique = snd (unique-walk-from-r-to b) _

      𝕨p·𝕖e≡𝕨q : (𝕨p ∙ 𝕖 e) ≡ 𝕨q
      𝕨p·𝕖e≡𝕨q =  𝕨 G (walk p) ∙ 𝕖 e
                ≡⟨ sym (comp-edge G (walk p) e) ⟩
                  𝕨 G ((_ , walk p , e))
                ≡⟨ cong (λ w → 𝕨 G (walk w)) (sym q-is-unique) ⟩
                  𝕨 G (walk q) ∎
```

## 6 Concluding Remarks

Here we present one short example of transferring some concepts and results from graph theory in a classical setting to Cubical type theory. As part of this process, we have used a proof assistant to support this goal. Precisely, we have characterised the notion of rooted trees to construct oriented spanning trees for directed multigraphs. These concepts are the generalisation of the notion of a tree and spanning tree for undirected graphs, respectively. A proof is given for the existence of an oriented rooted spanning tree for any strongly connected graph with a finite node set and a family of edges consisting of sets. To this end, we introduce a few lemmas that suggest algorithms for constructing spanning trees. Furthermore, we show that any rooted tree is a tree in the topological sense, inspired by Swan's work on defining free groups in HoTT and using higher—inductive types to model the topological realisation of graphs [9]. The results here can then be used to study free groups, particularly the fundamental group of a graph. In this direction, the realisation of a graph maps any of its spanning trees to a point in the space, and the remaining edges not in such a tree, become loops around the point. The loop edges then correspond to the elements of the group associated with the graph, sometimes called the fundamental group. We left this investigation for future work.

Most results here are formalised in Agda [10]. Except for proofs in Section 5, we conjecture it is only required intensional Martin–Löf type theory equipped with universes, function extensionality, and propositional truncation. To ease the work with higher–inductive types, especially in Section 5, we used the Cubical mode [12] in Agda and the Cubical Agda library [5]. Nevertheless, the type theory as presented in the HoTT Book [11] suffices to prove the results in this appendix.

Even when graph theory has been formalised before in type theory with proof-assistants, as the formalisation of the 4CT in Coq [2], there are still a few works in homotopy type theory [3, 4, 6]. As far as we know, the proofs and some types given here are original in this context. We believe this development contributes to the project of this thesis and the formalisation of mathematical content in type theories alike. We expect more contributions in this direction in the future.

A notable work close to ours is the recent work in Agda–UniMath [8], an Agda library for Univalent mathematics. Their authors formalised the notion of trees, rooted and quasi–rooted trees, for the case of undirected graphs. In future, we plan to transfer the results shown here to Agda–UniMath and make them available to a broader audience. In addition, ongoing work explores other topics, such as the two-cell realisation of a graph, where 2-cells correspond to faces [7] of a graph embedding.

## References


[1] Reinhard Diestel and Daniela Kühn. 2004. Topological paths, cycles and spanning trees in infinite graphs. *European Journal of Combinatorics* 25, 6 (Aug. 2004), 835–862. https://doi.org/10.1016/j.ejc.2003.01.002

[2] Georges Gonthier. 2008. Formal proof–the four-color theorem. *Notices of the AMS* 55, 11 (2008), 1382–1393. https://doi.org/10.1.1.141.714

[3] Nicolai Kraus and Jakob von Raumer. 2020. Coherence via Well-Foundedness. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. Acm. https://doi.org/10.1145/3373718.3394800

[4] Nicolai Kraus and Jakob von Raumer. 2021. A Rewriting Coherence Theorem with Applications in Homotopy Type Theory. arXiv:2107.01594 [cs.LO]

[5] Anders Mörtberg, Vezzosi Andrea, and Cavallo Evan. 2021. A Standard library for Cubical Agda. https://github.com/agda/cubical

[6] Jonathan Prieto-Cubides. 2022. On Homotopy of Walks and Spherical Maps in Homotopy Type Theory. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA) *(CPP 2022)*. Association for Computing Machinery, New York, NY, USA, 338–351. https://doi.org/10.1145/3497775.3503671

[7] Jonathan Prieto-Cubides and Håkon Robbestad Gylterud. 2022. On Planarity of Graphs in Homotopy Type Theory. (2022). arXiv:1601.05035 [cs.LO] https://arxiv.org/abs/ Submitted to *Mathematical Structures in Computer Science*.

[8] Egbert Rijke, Elisabeth Bonnevier, Jonathan Prieto-Cubides, et al. 2022. Univalent mathematics in Agda. https://unimath.github.io/agda-unimath/. (2022). https://doi.org/10.1007/snnnnn-nnnnn

[9] Andrew W Swan. 2022. On the Nielsen-Schreier Theorem in Homotopy Type Theory. *Logical Methods in Computer Science* Volume 18, Issue 1 (jan 2022). https://doi.org/10.46298/lmcs-18(1:18)2022

[10] The Agda Development Team. 2021. Agda 2.6.1.3 documentation. https://agda.readthedocs.io/en/v2.6.1.3/

[11] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

[12] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2021. Cubical Agda: A dependently typed programming language with univalence and higher inductive types . *Journal of Functional Programming* 31 (2021), e8. https://doi.org/10.1017/s0956796821000034