A Probabilistic Proof of the Girth-Chromatic Number Theorem

Lars Noschinski

March 14, 2025

Abstract

This works presents a formalization of the Girth-Chromatic number theorem in graph theory, stating that graphs with arbitrarily large girth and chromatic number exist. The proof uses the theory of Random Graphs to prove the existence with probabilistic arguments and is based on [1].

Contents

theory *Girth-Chromatic-Misc*
imports
Main
HOL−Library.Extended-Real
begin

1 Auxilliary lemmas and setup

This section contains facts about general concepts which are not directly connected to the proof of the Chromatic-Girth theorem. At some point in time, most of them could be moved to the Isabelle base library.

Also, a little bit of setup happens.

1.1 Numbers

lemma *enat-in-Inf*:
  fixes S :: *enat set*
  assumes *Inf* $S \neq$ *top*
  shows *Inf* $S \in S$
  using *assms wellorder-InfI* **by** *auto*

lemma *enat-in-INF*:
  fixes f :: $'a \Rightarrow$ *enat*
  assumes $(INF\ x{\in}S.\ f\ x) \neq top$
  obtains x **where** $x \in S$ **and** $(INF\ x{\in}S.\ f\ x) = f\ x$
  by (*meson assms enat-in-Inf imageE*)

lemma *enat-less-INF-I*:
  fixes f :: $'a \Rightarrow$ *enat*
  assumes *not-inf*: $x \neq \infty$ **and** *less*: $\bigwedge y.\ y \in S \Longrightarrow x < f\ y$
  shows $x < (INF\ y{\in}S.\ f\ y)$
  using *assms* **by** (*auto simp*: *Suc-ile-eq*[*symmetric*] *INF-greatest*)

lemma *enat-le-Sup-iff*:
  $enat\ k \leq Sup\ M \longleftrightarrow k = 0 \vee (\exists\, m \in M.\ enat\ k \leq m)$ (**is** *?L* $\longleftrightarrow$ *?R*)
proof *cases*
  assume $k = 0$ **then show** *?thesis* **by** (*auto simp*: *enat-0*)
next
  assume $k \neq 0$
  show *?thesis*
  proof
    assume *?L*
    then have $[\![enat\ k \leq (if\ finite\ M\ then\ Max\ M\ else\ \infty);\ M \neq \{\}]\!] \Longrightarrow \exists\, m{\in}M.$ *enat* $k \leq m$
      by (*metis Max-in Sup-enat-def finite-enat-bounded linorder-linear*)
    with ‹$k \neq 0$› **and** ‹*?L*› **show** *?R*
      unfolding *Sup-enat-def*
      by (*cases* $M{=}\{\}$) (*auto simp add*: *enat-0*[*symmetric*])
  next
    assume *?R* **then show** *?L*
      by (*auto simp*: *enat-0 intro*: *complete-lattice-class.Sup-upper2*)
  qed
qed

lemma *enat-neq-zero-cancel-iff*[*simp*]:
  $0 \neq enat\ n \longleftrightarrow 0 \neq n$
  $enat\ n \neq 0 \longleftrightarrow n \neq 0$
by (*auto simp*: *enat-0*[*symmetric*])

lemma *natceiling-lessD*: $nat(ceiling\ x) < n \Longrightarrow x < real\ n$
  by *linarith*

lemma *le-natceiling-iff*:
  fixes $n :: nat$ **and** $r :: real$
  shows $n \leq r \Longrightarrow n \leq nat(ceiling\ r)$
  by *linarith*

lemma *natceiling-le-iff*:
  fixes $n :: nat$ **and** $r :: real$
  shows $r \leq n \Longrightarrow nat(ceiling\ r) \leq n$
  by *linarith*

lemma *dist-real-noabs-less*:
  fixes $a\ b\ c :: real$ **assumes** $dist\ a\ b < c$ **shows** $a - b < c$
using *assms* **by** (*simp add*: *dist-real-def*)

1.2 Lists and Sets

lemma *list-set-tl*: $x \in set\ (tl\ xs) \Longrightarrow x \in set\ xs$
  by (*cases xs*) *auto*

lemma *list-exhaust3*:
  obtains $xs = [] \mid x$ **where** $xs = [x] \mid x\ y\ ys$ **where** $xs = x \# y \# ys$
  by (*metis list.exhaust*)

lemma *card-Ex-subset*:
  $k \leq card\ M \Longrightarrow \exists N.\ N \subseteq M \wedge card\ N = k$
  by (*induct rule*: *inc-induct*) (*auto simp*: *card-Suc-eq*)

1.3 Limits and eventually

We employ filters and the *eventually* predicate to deal with the $\exists N.\ \forall n{\geq}N.\ P\ n$ cases. To make this more convenient, introduce a shorter syntax.

abbreviation *evseq* :: $(nat \Rightarrow bool) \Rightarrow bool$ (**binder** $\langle\forall^{\infty}\rangle$ *10*) **where**
  $evseq\ P \equiv eventually\ P\ sequentially$

lemma *LIMSEQ-neg-powr*:
  assumes *s*: $s < 0$
  shows $(\%x.\ (real\ x)\ powr\ s) \longrightarrow 0$
  by (*simp add*: *filterlim-real-sequentially s tendsto-neg-powr*)

lemma *LIMSEQ-inv-powr*:

assumes $0 < c\ 0 < d$
shows $(\lambda n :: nat.\ (c\ /\ n)\ powr\ d) \longrightarrow 0$
proof (*rule tendsto-zero-powrI*)
from ‹$0 < c$› **have** $\bigwedge x.\ 0 < x \Longrightarrow 0 < c\ /\ x$ **by** *simp*
then show $\forall^{\infty} n.\ 0 \leq c\ /\ real\ n$
using *assms*(*1*) **by** *auto*
show $(\lambda x.\ c\ /\ real\ x) \longrightarrow 0$
by (*simp add*: *lim-const-over-n*)
qed (*use assms* **in** *force*)+

end
theory *Ugraphs*
imports
Girth-Chromatic-Misc
begin

2 Undirected Simple Graphs

In this section, we define some basics of graph theory needed to formalize the Chromatic-Girth theorem.

For readability, we introduce synonyms for the types of vertexes, edges, graphs and walks.

type-synonym *uvert* = *nat*
type-synonym *uedge* = *nat set*
type-synonym *ugraph* = *uvert set* × *uedge set*
type-synonym *uwalk* = *uvert list*

abbreviation *uedges* :: *ugraph* ⇒ *uedge set* **where**
uedges G ≡ *snd G*

abbreviation *uverts* :: *ugraph* ⇒ *uvert set* **where**
uverts G ≡ *fst G*

fun *mk-uedge* :: *uvert* × *uvert* ⇒ *uedge* **where**
mk-uedge (*u*,*v*) = {*u*,*v*}

All edges over a set of vertexes S:

definition *all-edges S* ≡ *mk-uedge* ‘ {*uv* ∈ *S* × *S*. *fst uv* ≠ *snd uv*}

definition *uwellformed* :: *ugraph* ⇒ *bool* **where**
uwellformed G ≡ (∀ *e*∈*uedges G*. *card e* = *2* ∧ (∀ *u* ∈ *e*. *u* ∈ *uverts G*))

fun *uwalk-edges* :: *uwalk* ⇒ *uedge list* **where**
uwalk-edges [] = []
| *uwalk-edges* [*x*] = []
| *uwalk-edges* (*x* # *y* # *ys*) = {*x*,*y*} # *uwalk-edges* (*y* # *ys*)

definition *uwalk-length* :: *uwalk* ⇒ *nat* **where**
uwalk-length p ≡ *length* (*uwalk-edges p*)

definition *uwalks* :: *ugraph* ⇒ *uwalk set* **where**
uwalks G ≡ {*p*. *set p* ⊆ *uverts G* ∧ *set* (*uwalk-edges p*) ⊆ *uedges G* ∧ *p* ≠ []}

definition *ucycles* :: *ugraph* ⇒ *uwalk set* **where**
ucycles G ≡ {*p*. *uwalk-length p* ≥ *3* ∧ *p* ∈ *uwalks G* ∧ *distinct* (*tl p*) ∧ *hd p* = *last p*}

definition *remove-vertex* :: *ugraph* ⇒ *nat* ⇒ *ugraph* (‹- −− -› [*60*,*60*] *60*) **where**
remove-vertex G u ≡ (*uverts G* − {*u*}, *uedges G* − {*A* ∈ *uedges G*. *u* ∈ *A*})

2.1 Basic Properties

lemma *uwalk-length-conv*: *uwalk-length p* = *length p* − *1*
by (*induct p rule*: *uwalk-edges.induct*) (*auto simp*: *uwalk-length-def*)

lemma *all-edges-mono*:
vs ⊆ *ws* ⟹ *all-edges vs* ⊆ *all-edges ws*
unfolding *all-edges-def* **by** *auto*

lemma *all-edges-subset-Pow*: *all-edges A* ⊆ *Pow A*
by (*auto simp*: *all-edges-def*)

lemma *in-mk-uedge-img*: (*a*,*b*) ∈ *A* ∨ (*b*,*a*) ∈ *A* ⟹ {*a*,*b*} ∈ *mk-uedge* ‘ *A*
by (*auto intro*: *rev-image-eqI*)

lemma *in-mk-uedge-img-iff*: {*a*,*b*} ∈ *mk-uedge* ‘ *A* ⟷ (*a*,*b*) ∈ *A* ∨ (*b*,*a*) ∈ *A*
by (*auto simp*: *doubleton-eq-iff intro*: *rev-image-eqI*)

lemma *distinct-edgesI*:
assumes *distinct p* **shows** *distinct* (*uwalk-edges p*)
proof −
from *assms* **have** *?thesis* ⋀*u*. *u* ∉ *set p* ⟹ (⋀*v*. *u* ≠ *v* ⟹ {*u*,*v*} ∉ *set* (*uwalk-edges p*))
by (*induct p rule*: *uwalk-edges.induct*) *auto*
then show *?thesis* **by** *simp*
qed

lemma *finite-ucycles*:
assumes *finite* (*uverts G*)
shows *finite* (*ucycles G*)
proof −
have *ucycles G* ⊆ {*xs*. *set xs* ⊆ *uverts G* ∧ *length xs* ≤ *Suc* (*card* (*uverts G*))}
proof (*rule*, *simp*)
fix *p* **assume** *p* ∈ *ucycles G*
then have *distinct* (*tl p*) **and** *set p* ⊆ *uverts G*
unfolding *ucycles-def uwalks-def* **by** *auto*

```
    moreover
    then have set (tl p) ⊆ uverts G
      by (auto simp: list-set-tl)
    with assms have card (set (tl p)) ≤ card (uverts G)
      by (rule card-mono)
    then have length (p) ≤ 1 + card (uverts G)
      using distinct-card[OF ‹distinct (tl p)›] by auto
   ultimately show set p ⊆ uverts G ∧ length p ≤ Suc (card (uverts G)) by auto
  qed
  moreover
  have finite {xs. set xs ⊆ uverts G ∧ length xs ≤ Suc (card (uverts G))}
    using assms by (rule finite-lists-length-le)
  ultimately
  show ?thesis by (rule finite-subset)
qed

lemma ucycles-distinct-edges:
  assumes c ∈ ucycles G shows distinct (uwalk-edges c)
proof −
  from assms have c-props: distinct (tl c) 4 ≤ length c hd c = last c
    by (auto simp add: ucycles-def uwalk-length-conv)
  then have {hd c, hd (tl c)} ∉ set (uwalk-edges (tl c))
  proof (induct c rule: uwalk-edges.induct)
    case (3 x y ys)
    then have hd ys ≠ last ys by (cases ys) auto
    moreover
    from 3 have uwalk-edges (y # ys) = {y, hd ys} # uwalk-edges ys
      by (cases ys) auto
    moreover
    { fix xs have set (uwalk-edges xs) ⊆ Pow (set xs)
        by (induct xs rule: uwalk-edges.induct) auto }
    ultimately
    show ?case using 3 by auto
  qed simp-all
  moreover
  from assms have distinct (uwalk-edges (tl c))
    by (intro distinct-edgesI) (simp add: ucycles-def)
  ultimately
  show ?thesis by (cases c rule: list-exhaust3) auto
qed

lemma card-left-less-pair:
  fixes A :: ('a :: linorder) set
  assumes finite A
  shows card {(a,b). a ∈ A ∧ b ∈ A ∧ a < b}
    = (card A * (card A − 1)) div 2
using assms
proof (induct A)
  case (insert x A)
```

```
  show ?case
  proof (cases card A)
    case (Suc n)
    have {(a,b). a ∈ insert x A ∧ b ∈ insert x A ∧ a < b}
       = {(a,b). a ∈ A ∧ b ∈ A ∧ a < b} ∪ (λa. if a < x then (a,x) else (x,a)) ‘ A
      using ‹x ∉ A› by (auto simp: order-less-le)
    moreover
    have finite {(a,b). a ∈ A ∧ b ∈ A ∧ a < b}
      using insert by (auto intro: finite-subset[of - A × A])
    moreover
    have {(a,b). a ∈ A ∧ b ∈ A ∧ a < b} ∩ (λa. if a < x then (a,x) else (x,a)) ‘
A = {}
      using ‹x ∉ A› by auto
    moreover have inj-on (λa. if a < x then (a, x) else (x, a)) A
      by (auto intro: inj-onI split: if-split-asm)
    ultimately show ?thesis using insert Suc
      by (simp add: card-Un-disjoint card-image del: if-image-distrib)
  qed (simp add: card-eq-0-iff insert)
qed simp

lemma card-all-edges:
  assumes finite A
  shows card (all-edges A) = card A choose 2
proof −
  have inj-on-mk-uedge: inj-on mk-uedge {(a,b). a < b}
    by (rule inj-onI) (auto simp: doubleton-eq-iff)
  have all-edges A = mk-uedge ‘ {(a,b). a ∈ A ∧ b ∈ A ∧ a < b} (is ?L = ?R)
    by (auto simp: all-edges-def intro!: in-mk-uedge-img)
  then have card ?L = card ?R by simp
  also have ... = card {(a,b). a ∈ A ∧ b ∈ A ∧ a < b}
    using inj-on-mk-uedge by (blast intro: card-image subset-inj-on)
  also have ... = (card A ∗ (card A − 1)) div 2
    using card-left-less-pair using assms by simp
  also have ... = (card A choose 2)
    by (simp add: choose-two)
  finally show ?thesis .
qed

lemma verts-Gu: uverts (G −− u) = uverts G − {u}
  unfolding remove-vertex-def by simp

lemma edges-Gu: uedges (G −− u) ⊆ uedges G
  unfolding remove-vertex-def by auto
```

2.2 Girth, Independence and Vertex Colorings

```
definition girth :: ugraph ⇒ enat where
  girth G ≡ INF p∈ ucycles G. enat (uwalk-length p)
```

definition *independent-sets* :: *ugraph* ⇒ *uvert set set* **where**
independent-sets Gr ≡ {*vs*. *vs* ⊆ *uverts Gr* ∧ *all-edges vs* ∩ *uedges Gr* = {}}

definition α :: *ugraph* ⇒ *enat* **where**
α *G* ≡ *SUP vs* ∈ *independent-sets G*. *enat* (*card vs*)

definition *vertex-colorings* :: *ugraph* ⇒ *uvert set set set* **where**
vertex-colorings G ≡ {*C*. ⋃ *C* = *uverts G* ∧ (∀ *c1*∈*C*. ∀ *c2*∈*C*. *c1* ≠ *c2* ⟶ *c1* ∩ *c2* = {}) ∧
(∀ *c*∈*C*. *c* ≠ {} ∧ (∀ *u* ∈ *c*. ∀ *v* ∈ *c*. {*u*,*v*} ∉ *uedges G*))}

The chromatic number χ:

definition *chromatic-number* :: *ugraph* ⇒ *enat* **where**
chromatic-number G ≡ *INF c*∈ (*vertex-colorings G*). *enat* (*card c*)

lemma *independent-sets-mono*:
vs ∈ *independent-sets G* ⟹ *us* ⊆ *vs* ⟹ *us* ∈ *independent-sets G*
using *Int-mono*[*OF all-edges-mono*, *of us vs uedges G uedges G*]
unfolding *independent-sets-def* **by** *auto*

lemma *le-α-iff*:
assumes *0* < *k*
shows *k* ≤ α *Gr* ⟷ *k* ∈ *card* ‘ *independent-sets Gr* (**is** *?L* ⟷ *?R*)
proof
assume *?L*
then obtain *vs* **where** *vs* ∈ *independent-sets Gr* **and** *k* ≤ *card vs*
using *assms* **unfolding** α*-def enat-le-Sup-iff* **by** *auto*
moreover
then obtain *us* **where** *us* ⊆ *vs* **and** *k* = *card us*
using *card-Ex-subset* **by** *auto*
ultimately
have *us* ∈ *independent-sets Gr* **by** (*auto intro*: *independent-sets-mono*)
then show *?R* **using** ‹*k* = *card us*› **by** *auto*
qed (*auto intro*: *SUP-upper simp*: α*-def*)

lemma *zero-less-α*:
assumes *uverts G* ≠ {}
shows *0* < α *G*
proof −
from *assms* **obtain** *a* **where** *a* ∈ *uverts G* **by** *auto*
then have *0* < *enat* (*card* {*a*}) {*a*} ∈ *independent-sets G*
by (*auto simp*: *independent-sets-def all-edges-def*)
then show *?thesis* **unfolding** α*-def less-SUP-iff* **..**
qed

lemma α*-le-card*:
assumes *finite* (*uverts G*)
shows α *G* ≤ *card*(*uverts G*)

proof −
  { fix x **assume** $x \in$ *independent-sets* G
    then have $x \subseteq$ *uverts* G **by** (*auto simp*: *independent-sets-def*) **}**
  with *assms* **show** *?thesis* **unfolding** α*-def*
    by (*intro SUP-least*) (*auto intro*: *card-mono*)
qed

lemma α*-fin*: *finite* (*uverts* G) $\Longrightarrow$ α $G \neq \infty$
  using α*-le-card*[*of* G] **by** (*cases* α G) *auto*

lemma α*-remove-le*:
  shows α (G −− u) $\leq$ α G
proof −
  have *independent-sets* (G −− u) $\subseteq$ *independent-sets* G (**is** *?L* $\subseteq$ *?R*)
   using *all-edges-subset-Pow* **by** (*simp add*: *independent-sets-def remove-vertex-def*)
blast
  then show *?thesis* **unfolding** α*-def*
    by (*rule SUP-subset-mono*) *simp*
qed

A lower bound for the chromatic number of a graph can be given in terms of the independence number

lemma *chromatic-lb*:
  assumes *wf-G*: *uwellformed* G
    and *fin-G*: *finite* (*uverts* G)
    and *neG*: *uverts* $G \neq \{\}$
  shows *card* (*uverts* G) / α $G \leq$ *chromatic-number* G
proof −
  from *wf-G* **have** ($\lambda v.\ \{v\}$) ' *uverts* $G \in$ *vertex-colorings* G
    by (*auto simp*: *vertex-colorings-def uwellformed-def*)
  then have *chromatic-number* $G \neq$ *top*
    by (*simp add*: *chromatic-number-def*) (*auto simp*: *top-enat-def*)
  then obtain *vc* **where** *vc-vc*: *vc* $\in$ *vertex-colorings* G
    and *vc-size*:*chromatic-number* G = *card vc*
    unfolding *chromatic-number-def* **by** (*rule enat-in-INF*)

  have *fin-vc-elems*: $\bigwedge c.\ c \in vc \Longrightarrow$ *finite* c
   using *vc-vc* **by** (*intro finite-subset*[*OF - fin-G*]) (*auto simp*: *vertex-colorings-def*)

  have *sum-vc-card*: ($\sum c \in vc.$ *card* c) = *card* (*uverts* G)
      using *fin-vc-elems vc-vc* **unfolding** *vertex-colorings-def*
      by (*simp add*: *card-Union-disjoint*[*symmetric*] *pairwise-def disjnt-def*)

  have $\bigwedge c.\ c \in vc \Longrightarrow c \in$ *independent-sets* G
   using *vc-vc* **by** (*auto simp*: *vertex-colorings-def independent-sets-def all-edges-def*)
  then have $\bigwedge c.\ c \in vc \Longrightarrow$ *card* $c \leq \alpha$ G
   using *vc-vc fin-vc-elems* **by** (*subst le-*α*-iff*) (*auto simp add*: *vertex-colorings-def*)
  then have ($\sum c \in vc.$ *card* c) $\leq$ *card vc* $*$ α G
    using *sum-bounded-above*[*of vc card* α G]

by (*simp add*: *of-nat-eq-enat*[*symmetric*] *of-nat-sum*)
then have *ereal-of-enat* (*card* (*uverts G*)) ≤ *ereal-of-enat* (*α G*) ∗ *ereal-of-enat* (*card vc*)
by (*simp add*: *sum-vc-card ereal-of-enat-pushout ac-simps del*: *ereal-of-enat-simps*)
with *zero-less-α*[*OF neG*] *α-fin*[*OF fin-G*] *vc-size* **show** *?thesis*
by (*simp add*: *ereal-divide-le-pos*)
qed

end
theory *Girth-Chromatic*
imports
Ugraphs
Girth-Chromatic-Misc
HOL−Probability.Probability
HOL−Decision-Procs.Approximation
begin

3 Probability Space on Sets of Edges

definition *cylinder* :: *'a set* ⇒ *'a set* ⇒ *'a set set* **where**
cylinder S A B = {*T* ∈ *Pow S*. *A* ⊆ *T* ∧ *B* ∩ *T* = {}}

lemma *full-sum*:
fixes *p* :: *real*
assumes *finite S*
shows (∑ *A*∈*Pow S*. *p*^*card A* ∗ (1 − *p*)^*card* (*S* − *A*)) = 1
using *assms*
proof *induct*
case (*insert s S*)
have *inj-on* (*insert s*) (*Pow S*)
and ⋀*x*. *S* − *insert s x* = *S* − *x*
and *Pow S* ∩ *insert s* ' *Pow S* = {}
and ⋀*x*. *x* ∈ *Pow S* ⟹ *card* (*insert s S* − *x*) = *Suc* (*card* (*S* − *x*))
using *insert*(*1−2*) **by** (*auto simp*: *insert-Diff-if intro*!: *inj-onI*)
moreover have ⋀*x*. *x* ⊆ *S* ⟹ *card* (*insert s x*) = *Suc* (*card x*)
using *insert*(*1−2*) **by** (*subst card.insert*) (*auto dest*: *finite-subset*)
ultimately show *?case*
by (*simp add*: *sum.reindex sum-distrib-left*[*symmetric*] *ac-simps*
insert.hyps sum.union-disjoint Pow-insert)
qed *simp*

Definition of the probability space on edges:

locale *edge-space* =
fixes *n* :: *nat* **and** *p* :: *real*
assumes *p-prob*: *0* ≤ *p p* ≤ *1*
begin

definition *S-verts* :: *nat set* **where**
S-verts ≡ {*1*..*n*}

definition *S-edges* :: *uedge set* **where**
S-edges = *all-edges S-verts*

definition *edge-ugraph* :: *uedge set* ⇒ *ugraph* **where**
edge-ugraph es ≡ (*S-verts*, *es* ∩ *S-edges*)

definition *P* = *point-measure* (*Pow S-edges*) (λ*s*. *p*^*card s* ∗ (*1* − *p*)^*card* (*S-edges* − *s*))

lemma *finite-verts*[*intro*!]: *finite S-verts*
by (*auto simp*: *S-verts-def*)

lemma *finite-edges*[*intro*!]: *finite S-edges*
by (*auto simp*: *S-edges-def all-edges-def finite-verts*)

lemma *finite-graph*[*intro*!]: *finite* (*uverts* (*edge-ugraph es*))
unfolding *edge-ugraph-def* **by** *auto*

lemma *uverts-edge-ugraph*[*simp*]: *uverts* (*edge-ugraph es*) = *S-verts*
by (*simp add*: *edge-ugraph-def*)

lemma *uedges-edge-ugraph*[*simp*]: *uedges* (*edge-ugraph es*) = *es* ∩ *S-edges*
unfolding *edge-ugraph-def* **by** *simp*

lemma *space-eq*: *space P* = *Pow S-edges* **by** (*simp add*: *P-def space-point-measure*)

lemma *sets-eq*: *sets P* = *Pow* (*Pow S-edges*) **by** (*simp add*: *P-def sets-point-measure*)

lemma *emeasure-eq*:
emeasure P A = (*if A* ⊆ *Pow S-edges then* (∑ *edges*∈*A*. *p*^*card edges* ∗ (*1* − *p*)^*card* (*S-edges* − *edges*)) *else 0*)
using *finite-edges p-prob*
by (*simp add*: *P-def space-point-measure emeasure-point-measure-finite*
sets-point-measure emeasure-notin-sets)

lemma *integrable-P*[*intro*, *simp*]: *integrable P* (*f*::- ⇒ *real*)
using *finite-edges* **by** (*simp add*: *integrable-point-measure-finite P-def*)

lemma *borel-measurable-P*[*measurable*]: *f* ∈ *borel-measurable P*
unfolding *P-def* **by** *simp*

lemma *prob-space-P*: *prob-space P*
proof
show *emeasure P* (*space P*) = *1* — Sum of probabilities equals 1
using *finite-edges* **by** (*simp add*: *emeasure-eq full-sum one-ereal-def space-eq*)
qed

end

sublocale *edge-space* $\subseteq$ *prob-space P*
by (*rule prob-space-P*)

context *edge-space*
begin

lemma *prob-eq*:
prob A = (*if A* $\subseteq$ *Pow S-edges then* ($\sum$ *edges*$\in$*A. p*^*card edges* $*$ (*1* $-$ *p*)^*card* (*S-edges* $-$ *edges*)) *else 0*)
using *emeasure-eq*[*of A*] *p-prob* **unfolding** *emeasure-eq-measure* **by** (*simp add*: *sum-nonneg*)

lemma *integral-finite-singleton*: $integral^L$ *P f* = ($\sum$ *x*$\in$*Pow S-edges. f x* $*$ *measure P* {*x*})
using *p-prob prob-eq* **unfolding** *P-def*
by (*subst lebesgue-integral-point-measure-finite*) (*auto intro*!: *sum.cong*)

Probability of cylinder sets:

lemma *cylinder-prob*:
assumes *A* $\subseteq$ *S-edges B* $\subseteq$ *S-edges A* $\cap$ *B* = {}
shows *prob* (*cylinder S-edges A B*) = *p* ^ (*card A*) $*$ (*1* $-$ *p*) ^ (*card B*) (**is** - = *?pp A B*)
proof $-$
have *Pow S-edges* $\cap$ *cylinder S-edges A B* = *cylinder S-edges A B*
$\bigwedge$*x. x* $\in$ *cylinder S-edges A B* $\Longrightarrow$ *A* $\cup$ *x* = *x*
$\bigwedge$*x. x* $\in$ *cylinder S-edges A B* $\Longrightarrow$ *finite x*
$\bigwedge$*x. x* $\in$ *cylinder S-edges A B* $\Longrightarrow$ *B* $\cap$ (*S-edges* $-$ *B* $-$ *x*) = {}
$\bigwedge$*x. x* $\in$ *cylinder S-edges A B* $\Longrightarrow$ *B* $\cup$ (*S-edges* $-$ *B* $-$ *x*) = *S-edges* $-$ *x*
finite A finite B
using *assms* **by** (*auto simp add*: *cylinder-def intro*: *finite-subset*)
then have ($\sum$ *T*$\in$*cylinder S-edges A B. ?pp T* (*S-edges* $-$ *T*))
= ($\sum$ *T* $\in$ *cylinder S-edges A B. p*^(*card A* + *card* (*T* $-$ *A*)) $*$ (*1* $-$ *p*)^(*card B* + *card* ((*S-edges* $-$ *B*) $-$ *T*)))
using *finite-edges* **by** (*simp add*: *card-Un-Int*)
also have ... = *?pp A B* $*$ ($\sum$ *T*$\in$*cylinder S-edges A B. ?pp* (*T* $-$ *A*) (*S-edges* $-$ *B* $-$ *T*))
by (*simp add*: *power-add sum-distrib-left ac-simps*)
also have ... = *?pp A B*
proof $-$
have $\bigwedge$*T. T* $\in$ *cylinder S-edges A B* $\Longrightarrow$ *S-edges* $-$ *B* $-$ *T* = (*S-edges* $-$ *A*) $-$ *B* $-$ (*T* $-$ *A*)
Pow (*S-edges* $-$ *A* $-$ *B*) = (*λx. x* $-$ *A*) ‘ *cylinder S-edges A B*
inj-on (*λx. x* $-$ *A*) (*cylinder S-edges A B*)
finite (*S-edges* $-$ *A* $-$ *B*)
using *assms* **by** (*auto simp*: *cylinder-def intro*!: *inj-onI*)
with *full-sum*[*of S-edges* $-$ *A* $-$ *B*] **show** *?thesis* **by** (*simp add*: *sum.reindex*)
qed
finally show *?thesis* **by** (*auto simp add*: *prob-eq cylinder-def*)

qed

lemma *Markov-inequality*:
fixes $a :: real$ **and** $X :: uedge\ set \Rightarrow real$
assumes $0 < c\ \bigwedge x.\ 0 \leq f\ x$
shows $prob\ \{x \in space\ P.\ c \leq f\ x\} \leq (\int x.\ f\ x\ \partial\ P)\ /\ c$
proof −
from *assms* **have** $(\int^+ x.\ ennreal\ (f\ x)\ \partial P) = (\int x.\ f\ x\ \partial P)$
by (*intro nn-integral-eq-integral*) *auto*
with *assms* **show** *?thesis*
using *nn-integral-Markov-inequality*[*of f space P P 1 / c*]
by (*simp cong*: *nn-integral-cong add*: *emeasure-eq-measure ennreal-mult*[*symmetric*])
qed

end

3.1 Graph Probabilities outside of *Edge-Space* locale

These abbreviations allow a compact expression of probabilities about random graphs outside of the *Edge-Space* locale. We also transfer a few of the lemmas we need from the locale into the toplevel theory.

abbreviation $MGn :: (nat \Rightarrow real) \Rightarrow nat \Rightarrow (uedge\ set)\ measure$ **where**
$MGn\ p\ n \equiv (edge\text{-}space.P\ n\ (p\ n))$
abbreviation $probGn :: (nat \Rightarrow real) \Rightarrow nat \Rightarrow (uedge\ set \Rightarrow bool) \Rightarrow real$ **where**
$probGn\ p\ n\ P \equiv measure\ (MGn\ p\ n)\ \{es \in space\ (MGn\ p\ n).\ P\ es\}$

lemma *probGn-le*:
assumes *p-prob*: $0 < p\ n\ p\ n < 1$
assumes *sub*: $\bigwedge n\ es.\ es \in space\ (MGn\ p\ n) \Longrightarrow P\ n\ es \Longrightarrow Q\ n\ es$
shows $probGn\ p\ n\ (P\ n) \leq probGn\ p\ n\ (Q\ n)$
proof −
from *p-prob* **interpret** *E*: *edge-space n p n* **by** *unfold-locales auto*
show *?thesis*
by (*auto intro*!: *E.finite-measure-mono sub simp*: *E.space-eq E.sets-eq*)
qed

4 Short cycles

definition $short\text{-}cycles :: ugraph \Rightarrow nat \Rightarrow uwalk\ set$ **where**
$short\text{-}cycles\ G\ k \equiv \{p \in ucycles\ G.\ uwalk\text{-}length\ p \leq k\}$

obtains a vertex in a short cycle:

definition $choose\text{-}v :: ugraph \Rightarrow nat \Rightarrow uvert$ **where**
$choose\text{-}v\ G\ k \equiv SOME\ u.\ \exists p.\ p \in short\text{-}cycles\ G\ k \wedge u \in set\ p$

partial-function (*tailrec*) $kill\text{-}short :: ugraph \Rightarrow nat \Rightarrow ugraph$ **where**
$kill\text{-}short\ G\ k = (if\ short\text{-}cycles\ G\ k = \{\}\ then\ G\ else\ (kill\text{-}short\ (G\ --\ (choose\text{-}v\ G\ k))\ k))$

lemma *ksc-simps*[*simp*]:
short-cycles G k = {} ⟹ *kill-short G k* = *G*
short-cycles G k ≠ {} ⟹ *kill-short G k* = *kill-short* (*G* −− (*choose-v G k*)) *k*
by (*auto simp*: *kill-short.simps*)

lemma
assumes *short-cycles G k* ≠ {}
shows *choose-v--in-uverts*: *choose-v G k* ∈ *uverts G* (**is** *?t1*)
and *choose-v--in-short*: ∃ *p*. *p* ∈ *short-cycles G k* ∧ *choose-v G k* ∈ *set p* (**is** *?t2*)
proof −
from *assms* **obtain** *p* **where** *p* ∈ *ucycles G uwalk-length p* ≤ *k*
unfolding *short-cycles-def* **by** *auto*
moreover
then obtain *u* **where** *u* ∈ *set p* **unfolding** *ucycles-def*
by (*cases p*) (*auto simp*: *uwalk-length-conv*)
ultimately have ∃ *u p*. *p* ∈ *short-cycles G k* ∧ *u* ∈ *set p*
by (*auto simp*: *short-cycles-def*)
then show *?t2* **by** (*auto simp*: *choose-v-def intro*!: *someI-ex*)
then show *?t1* **by** (*auto simp*: *short-cycles-def ucycles-def uwalks-def*)
qed

lemma *kill-step-smaller*:
assumes *short-cycles G k* ≠ {}
shows *short-cycles* (*G* −− (*choose-v G k*)) *k* ⊂ *short-cycles G k*
proof −
let *?cv* = *choose-v G k*
from *assms* **obtain** *p* **where** *p* ∈ *short-cycles G k ?cv* ∈ *set p*
by *atomize-elim* (*rule choose-v--in-short*)

have *short-cycles* (*G* −− *?cv*) *k* ⊆ *short-cycles G k*
proof
fix *p* **assume** *p* ∈ *short-cycles* (*G* −− *?cv*) *k*
then show *p* ∈ *short-cycles G k*
unfolding *short-cycles-def ucycles-def uwalks-def*
using *edges-Gu*[*of G ?cv*] **by** (*auto simp*: *verts-Gu*)
qed
moreover have *p* ∉ *short-cycles* (*G* −− *?cv*) *k*
using ‹*?cv* ∈ *set p*› **by** (*auto simp*: *short-cycles-def ucycles-def uwalks-def verts-Gu*)
ultimately show *?thesis* **using** ‹*p* ∈ *short-cycles G k*› **by** *auto*
qed

Induction rule for *kill-short*:

lemma *kill-short-induct*[*consumes 1*, *case-names empty kill-vert*]:
assumes *fin*: *finite* (*uverts G*)
assumes *a-empty*: ⋀*G*. *short-cycles G k* = {} ⟹ *P G k*
assumes *a-kill*: ⋀*G*. *finite* (*short-cycles G k*) ⟹ *short-cycles G k* ≠ {}

```
    ⟹ P (G −− (choose-v G k)) k ⟹ P G k
  shows P G k
proof −
  have finite (short-cycles G k)
    using finite-ucycles[OF fin] by (auto simp: short-cycles-def)
  then show ?thesis
    by (induct short-cycles G k arbitrary: G rule: finite-psubset-induct)
      (metis kill-step-smaller a-kill a-empty)
qed
```

Large Girth (after *kill-short*):

```
lemma kill-short-large-girth:
  assumes finite (uverts G)
  shows k < girth (kill-short G k)
using assms
proof (induct G k rule: kill-short-induct)
  case (empty G)
  then have ⋀p. p ∈ ucycles G ⟹ k < enat (uwalk-length p)
    by (auto simp: short-cycles-def)
  with empty show ?case by (auto simp: girth-def intro: enat-less-INF-I)
qed simp
```

Order of graph (after *kill-short*):

```
lemma kill-short-order-of-graph:
  assumes finite (uverts G)
  shows card (uverts G) − card (short-cycles G k) ≤ card (uverts (kill-short G k))
using assms assms
proof (induct G k rule: kill-short-induct)
  case (kill-vert G)
  let ?oG = G −− (choose-v G k)

  have finite (uverts ?oG)
    using kill-vert by (auto simp: remove-vertex-def)
  moreover
  have uverts (kill-short G k) = uverts (kill-short ?oG k)
    using kill-vert by simp
  moreover
  have card (uverts G) = Suc (card (uverts ?oG))
    using choose-v--in-uverts kill-vert
    by (simp add: remove-vertex-def card-Suc-Diff1 del: card-Diff-insert)
  moreover
  have card (short-cycles ?oG k) < card (short-cycles G k)
    by (intro psubset-card-mono kill-vert.hyps kill-step-smaller)
  ultimately show ?case using kill-vert.hyps by presburger
qed simp
```

Independence number (after *kill-short*):

```
lemma kill-short-α:
  assumes finite (uverts G)
```

shows α (*kill-short G k*) $\leq \alpha$ *G*
using *assms*
proof (*induct G k rule*: *kill-short-induct*)
case (*kill-vert G*)
note *kill-vert*(*3*)
also have α (*G* −− (*choose-v G k*)) $\leq \alpha$ *G* **by** (*rule* α-*remove-le*)
finally show *?case* **using** *kill-vert* **by** *simp*
qed *simp*

Wellformedness (after *kill-short*):

lemma *kill-short-uwellformed*:
assumes *finite* (*uverts G*) *uwellformed G*
shows *uwellformed* (*kill-short G k*)
using *assms*
proof (*induct G k rule*: *kill-short-induct*)
case (*kill-vert G*)
from *kill-vert.prems* **have** *uwellformed* (*G* −− (*choose-v G k*))
by (*auto simp*: *uwellformed-def remove-vertex-def*)
with *kill-vert.hyps* **show** *?case* **by** *simp*
qed *simp*

5 The Chromatic-Girth Theorem

Probability of Independent Edges:

lemma (**in** *edge-space*) *random-prob-independent*:
assumes *n* ≥ *k k* ≥ *2*
shows *prob* {*es* ∈ *space P*. *k* ≤ α (*edge-ugraph es*)}
≤ (*n choose k*)∗(*1*−*p*)^(*k choose 2*)
proof −
let *?k-sets* = {*vs*. *vs* ⊆ *S-verts* ∧ *card vs* = *k*}

{ fix *vs* **assume** *A*: *vs* ∈ *?k-sets*
then have *B*: *all-edges vs* ⊆ *S-edges*
unfolding *all-edges-def S-edges-def* **by** *blast*

have {*es* ∈ *space P*. *vs* ∈ *independent-sets* (*edge-ugraph es*)}
= *cylinder S-edges* {} (*all-edges vs*) (**is** *?L* = -)
using *A* **by** (*auto simp*: *independent-sets-def edge-ugraph-def space-eq cylinder-def*)
then have *prob ?L* = (*1*−*p*)^(*k choose 2*)
using *A B finite* **by** (*auto simp*: *cylinder-prob card-all-edges dest*: *finite-subset*)
}
note *prob-k-indep* = *this*
— probability that a fixed set of k vertices is independent in a random graph

have {*es* ∈ *space P*. *k* ∈ *card* ‘ *independent-sets* (*edge-ugraph es*)}
= (⋃ *vs* ∈ *?k-sets*. {*es* ∈ *space P*. *vs* ∈ *independent-sets* (*edge-ugraph es*)}) (**is** *?L* = *?R*)

unfolding *image-def space-eq independent-sets-def* **by** *auto*
then have *prob ?L* $\leq$ ($\sum$ *vs* $\in$ *?k-sets. prob* {*es* $\in$ *space P. vs* $\in$ *independent-sets* (*edge-ugraph es*)})
by (*auto intro*!: *finite-measure-subadditive-finite simp*: *space-eq sets-eq*)
also have ... = (*n choose k*)*((1 − *p*) ^ (*k choose 2*))
by (*simp add*: *prob-k-indep S-verts-def n-subsets*)
finally show *?thesis* **using** ‹*k* $\geq$ *2*› **by** (*simp add*: *le-α-iff*)
qed

Almost never many independent edges:

lemma *almost-never-le-α*:
fixes *k* :: *nat*
and *p* :: *nat* ⇒ *real*
assumes *p-prob*: ∀$^\infty$ *n*. *0* < *p n* ∧ *p n* < *1*
assumes [*arith*]: *k* > *0*
assumes *N-prop*: ∀$^\infty$ *n*. (*6* ∗ *k* ∗ *ln n*)/*n* ≤ *p n*
shows (λ*n*. *probGn p n* (λ*es*. *1*/*2*∗*n*/*k* ≤ *α* (*edge-space.edge-ugraph n es*))) ⟶ *0*
(**is** (λ*n*. *?prob-fun n*) ⟶ *0*)
proof −
let *?prob-fun-raw n* = *probGn p n* (λ*es*. *nat*(*ceiling* (*1*/*2*∗*n*/*k*)) ≤ *α* (*edge-space.edge-ugraph n es*))

define *r* **where** *r n* = *1* / *2* ∗ *n* / *k* **for** *n* :: *nat*
let *?nr* = λ*n*. *nat*(*ceiling* (*r n*))

have *r-pos*: ⋀*n*. *0* < *n* ⟹ *0* < *r n* **by** (*auto simp*: *r-def field-simps*)

have *nr-bounds*: ∀$^\infty$ *n*. *2* ≤ *?nr n* ∧ *?nr n* ≤ *n*
by (*intro eventually-sequentiallyI*[*of 4* ∗ *k*])
(*simp add*: *r-def nat-ceiling-le-eq le-natceiling-iff field-simps*)

from *nr-bounds p-prob* **have** *ev-prob-fun-raw-le*:
∀$^\infty$ *n*. *probGn p n* (λ*es*. *?nr n*≤ *α* (*edge-space.edge-ugraph n es*))
≤ (*n* ∗ *exp* (− *p n* ∗ (*real* (*?nr n*) − *1*) / *2*)) *powr ?nr n*
(**is** ∀$^\infty$ *n*. *?prob-fun-raw-le n*)
proof (*rule eventually-elim2*)
fix *n* :: *nat* **assume** *A*: *2* ≤ *?nr n* ∧ *?nr n* ≤ *n 0* < *p n* ∧*p n* < *1*
then interpret *pG*: *edge-space n p n* **by** *unfold-locales auto*

have *r*: *real* (*?nr n* − *Suc 0*) = *real* (*?nr n*) − *Suc 0* **using** *A* **by** *auto*

have [*simp*]: *n*>*0* **using** *A* **by** *linarith*
have *probGn p n* (λ*es*. *?nr n* ≤ *α* (*edge-space.edge-ugraph n es*))
≤ (*n choose ?nr n*) ∗ (*1* − *p n*)^(*?nr n choose 2*)
using *A* **by** (*auto intro*: *pG.random-prob-independent*)
also have ... ≤ *n powr ?nr n* ∗ (*1* − *p n*) *powr* (*?nr n choose 2*)
using *A* **by** (*simp add*: *powr-realpow of-nat-power* [*symmetric*] *binomial-le-pow del*: *of-nat-power*)

```
    also have ... = n powr ?nr n * (1 - p n) powr (?nr n * (?nr n - 1) / 2)
      by (cases even (?nr n - 1))
        (auto simp add: choose-two real-of-nat-div)
    also have ... = n powr ?nr n * ((1 - p n) powr ((?nr n - 1) / 2)) powr ?nr
n
      by (auto simp add: powr-powr r ac-simps)
    also have ... ≤ (n * exp (- p n * (?nr n - 1) / 2)) powr ?nr n
    proof -
      have (1 - p n) powr ((?nr n - 1) / 2) ≤ exp (- p n) powr ((?nr n - 1) /
2)
          using A by (auto simp: powr-mono2 diff-conv-add-uminus simp del:
add-uminus-conv-diff)
      also have ... = exp (- p n * (?nr n - 1) / 2) by (auto simp: powr-def)
      finally show ?thesis
        using A by (auto simp: powr-mono2 powr-mult)
    qed
    finally show probGn p n (λes. ?nr n ≤ α (edge-space.edge-ugraph n es))
      ≤ (n * exp (- p n * (real (?nr n) - 1) / 2)) powr ?nr n
      using A r by simp
  qed

  from p-prob N-prop
  have ev-expr-bound: ∀∞ n. n * exp (-p n * (real (?nr n) - 1) / 2) ≤ (exp 1 /
n) powr (1 / 2)
  proof (elim eventually-rev-mp, intro eventually-sequentiallyI conjI impI)
    fix n assume n-bound[arith]: 2 ≤ n
      and p-bound: 0 < p n ∧ p n < 1 (6 * k * ln n) / n ≤ p n
    have r-bound: r n ≤ ?nr n by (rule real-nat-ceiling-ge)

    have n * exp (-p n * (real (?nr n)- 1) / 2) ≤ n * exp (- 3 / 2 * ln n + p
n / 2)
    proof -
      have 0 < ln n using n-bound by auto
      then have (3 / 2) * ln n ≤ ((6 * k * ln n) / n) * (?nr n / 2)
        using r-bound le-of-int-ceiling[of n/2*k]
        by (simp add: r-def field-simps del: le-of-int-ceiling)
      also have ... ≤ p n * (?nr n / 2)
        using n-bound p-bound r-bound r-pos[of n] by (auto simp: field-simps)
      finally show ?thesis using r-bound by (auto simp: field-simps)
    qed
    also have ... ≤ n * n powr (- 3 / 2) * exp 1 powr (1 / 2)
      using p-bound by (simp add: powr-def exp-add [symmetric])
     also have ... ≤ n powr (-1 / 2) * exp 1 powr (1 / 2) by (simp add:
powr-mult-base)
    also have ... = (exp 1 / n) powr (1/2)
      by (simp add: powr-divide powr-minus-divide)
    finally show n * exp (- p n * (real (?nr n) - 1) / 2) ≤ (exp 1 / n) powr (1
/ 2) .
  qed
```

```
  have ceil-bound: ⋀G n. 1/2*n/k ≤ α G ⟷ nat(ceiling (1/2*n/k)) ≤ α G
    by (case-tac α G) (auto simp: nat-ceiling-le-eq)

  show ?thesis
  proof (unfold ceil-bound, rule real-tendsto-sandwich)
    show (λn. 0) ⟶ 0
        (λn. (exp 1 / n) powr (1 / 2)) ⟶ 0
        ∀⁰⁰ n. 0 ≤ ?prob-fun-raw n
      using p-prob by (auto intro: measure-nonneg LIMSEQ-inv-powr elim: eventually-mono)
  next
    from nr-bounds ev-expr-bound ev-prob-fun-raw-le
    show ∀⁰⁰ n. ?prob-fun-raw n ≤ (exp 1 / n) powr (1 / 2)
    proof (elim eventually-rev-mp, intro eventually-sequentiallyI impI conjI)
      fix n assume A: 3 ≤ n
        and nr-bounds: 2 ≤ ?nr n ∧ ?nr n ≤ n
        and prob-fun-raw-le: ?prob-fun-raw-le n
        and expr-bound: n * exp (- p n * (real (nat(ceiling (r n))) - 1) / 2) ≤ (exp 1 / n) powr (1 / 2)

      have exp 1 < (3 :: real) by (approximation 6)
      then have (exp 1 / n) powr (1 / 2) ≤ 1 powr (1 / 2)
        using A by (intro powr-mono2) (auto simp: field-simps)
      then have ep-bound: (exp 1 / n) powr (1 / 2) ≤ 1 by simp

       have ?prob-fun-raw n ≤ (n * exp (- p n * (real (?nr n) - 1) / 2)) powr (?nr n)
        using prob-fun-raw-le by (simp add: r-def)
      also have ... ≤ ((exp 1 / n) powr (1 / 2)) powr ?nr n
        using expr-bound A by (auto simp: powr-mono2)
      also have ... ≤ ((exp 1 / n) powr (1 / 2))
        using nr-bounds ep-bound A by (auto simp: powr-le-one-le)
      finally show ?prob-fun-raw n ≤ (exp 1 / n) powr (1 / 2) .
    qed
  qed
qed
```

Mean number of k-cycles in a graph. (Or rather of paths describing a circle of length k):

```
lemma (in edge-space) mean-k-cycles:
  assumes 3 ≤ k k < n
  shows (∫ es. card {c ∈ ucycles (edge-ugraph es). uwalk-length c = k} ∂ P)
    = of-nat (fact n div fact (n - k)) * p ^ k
proof -
  let ?k-cycle = λes c k. c ∈ ucycles (edge-ugraph es) ∧ uwalk-length c = k
  define C where C k = {c. ?k-cycle S-edges c k} for k
    — C k is the set of all possible cycles of size k in edge-ugraph S-edges
  define XG  where XG es = {c. ?k-cycle es c k} for es
```

— *XG es* is the set of cycles contained in a *edge-ugraph es*
define *XC* **where** *XC c* = {*es* ∈ *space P*. *?k-cycle es c k*} **for** *c*
— "*XC c* is the set of graphs (edge sets) containing a cycle c"
then have *XC-in-sets*: ⋀*c*. *XC c* ∈ *sets P*
and *XC-cyl*: ⋀*c*. *c* ∈ *C k* ⟹ *XC c* = *cylinder S-edges* (*set* (*uwalk-edges c*)) {}
by (*auto simp*: *ucycles-def space-eq uwalks-def C-def cylinder-def sets-eq*)

have (∫ *es*. *card* {*c* ∈ *ucycles* (*edge-ugraph es*). *uwalk-length c* = *k*} ∂ *P*)
= (∑ *x*∈*space P*. *card* (*XG x*) ∗ *prob* {*x*})
by (*simp add*: *XG-def integral-finite-singleton space-eq*)
also have ... = (∑ *c*∈*C k*. *prob* (*cylinder S-edges* (*set* (*uwalk-edges c*)) {}))
proof −
have *XG-Int-C*: ⋀*s*. *s* ∈ *space P* ⟹ *C k* ∩ *XG s* = *XG s*
unfolding *XG-def C-def ucycles-def uwalks-def edge-ugraph-def* **by** *auto*
have *fin-XC*: ⋀*k*. *finite* (*XC k*) **and** *fin-C*: *finite* (*C k*)
unfolding *C-def XC-def* **by** (*auto simp*: *finite-edges space-eq intro*!: *finite-ucycles*)

have (∑ *x*∈*space P*. *card* (*XG x*) ∗ *prob* {*x*}) = (∑ *x*∈*space P*. (∑ *c* ∈ *XG x*. *prob* {*x*}))
by *simp*
also have ... = (∑ *x*∈*space P*. (∑ *c* ∈ *C k*. *if c* ∈ *XG x then prob* {*x*} *else* 0))
using *fin-C* **by** (*simp add*: *sum.If-cases*) (*simp add*: *XG-Int-C*)
also have ... = (∑ *c* ∈ *C k*. (∑ *x* ∈ *space P* ∩ *XC c*. *prob* {*x*}))
using *finite-edges* **by** (*subst sum.swap*) (*simp add*: *sum.inter-restrict XG-def XC-def space-eq*)
also have ... = (∑ *c* ∈ *C k*. *prob* (*XC c*))
using *fin-XC XC-in-sets*
by (*auto simp add*: *prob-eq sets-eq space-eq intro*!: *sum.cong*)
finally show *?thesis* **by** (*simp add*: *XC-cyl*)
qed
also have ... = (∑ *c*∈*C k*. *p* ^ *k*)
proof −
have ⋀*x*. *x* ∈ *C k* ⟹ *card* (*set* (*uwalk-edges x*)) = *uwalk-length x*
by (*auto simp*: *uwalk-length-def C-def ucycles-distinct-edges intro*: *distinct-card*)
then show *?thesis* **by** (*auto simp*: *C-def ucycles-def uwalks-def cylinder-prob*)
qed
also have ... = *of-nat* (*fact n div fact* (*n* − *k*)) ∗ *p* ^ *k*
proof −
have *inj-last-Cons*: ⋀*A*. *inj-on* (λ*es*. *last es* # *es*) *A* **by** (*rule inj-onI*) *simp*
{ fix *xs A* **assume** *3* ≤ *length xs* − *Suc 0 hd xs* = *last xs*
then have *xs* ∈ (λ*xs*. *last xs* # *xs*) ‘ *A* ⟷ *tl xs* ∈ *A*
by (*cases xs*) (*auto simp*: *inj-image-mem-iff*[*OF inj-last-Cons*] *split*: *if-split-asm*)
}
note *image-mem-iff-inst* = *this*

{ fix *xs* **have** *xs* ∈ *uwalks* (*edge-ugraph S-edges*) ⟹ *set* (*tl xs*) ⊆ *S-verts*

```
      unfolding uwalks-def by (induct xs) auto }
    moreover
    { fix xs assume set xs ⊆ S-verts 2 ≤ length xs distinct xs
      then have (last xs # xs) ∈ uwalks (edge-ugraph S-edges)
      proof (induct xs rule: uwalk-edges.induct)
        case (3 x y ys)
        have S-edges-memI: ⋀x y. x ∈ S-verts ⟹ y ∈ S-verts ⟹ x ≠ y ⟹ {x,
y} ∈ S-edges
          unfolding S-edges-def all-edges-def image-def by auto

        have ys ≠ [] ⟹ set ys ⊆ S-verts ⟹ last ys ∈ S-verts  by auto
        with 3 show ?case
          by (auto simp add: uwalks-def Suc-le-eq intro: S-edges-memI)
      qed simp-all}
    moreover note ‹3 ≤ k›
    ultimately
    have C k = (λxs. last xs # xs) ` {xs. length xs = k ∧ distinct xs ∧ set xs ⊆
S-verts}
      by (auto simp: C-def ucycles-def uwalk-length-conv image-mem-iff-inst)
    moreover have card S-verts = n by (simp add: S-verts-def)
    ultimately have card (C k) = fact n div fact (n − k)
      using ‹k < n›
    by (simp add: card-image[OF inj-last-Cons] card-lists-distinct-length-eq' fact-div-fact)
    then show ?thesis by simp
  qed
  finally show ?thesis by simp
qed
```

Girth-Chromatic number theorem:

```
theorem girth-chromatic:
  fixes l :: nat
  shows ∃ G. uwellformed G ∧ l < girth G ∧ l < chromatic-number G
proof −
  define k where k = max 3 l
  define ε where ε = 1 / (2 * k)
  define p where p n = real n powr (ε − 1) for n :: nat

  let ?ug = edge-space.edge-ugraph

  define short-count where short-count g = card (short-cycles g k) for g
    — This random variable differs from the one used in the proof of theorem 11.2.2,
as we count the number of paths describing a circle, not the circles themselves

  from k-def have 3 ≤ k l ≤ k by auto
   from ‹3 ≤ k› have ε-props: 0 < ε ε < 1 / k ε < 1 by (auto simp: ε-def
field-simps)

  have ev-p: ∀∞ n. 0 < p n ∧ p n < 1
  proof (rule eventually-sequentiallyI)
```

```
    fix n :: nat assume 2 ≤ n
    with ‹ε < 1› have n powr (ε − 1) < 1 by (auto intro!: powr-less-one)
    then show 0 < p n ∧ p n < 1 using ‹2 ≤ n›
      by (auto simp: p-def)
  qed
  then
  have prob-short-count-le: ∀∞ n. probGn p n (λes. (real n/2) ≤ short-count (?ug
n es))
      ≤ 2 * (k − 2) * n powr (ε * k − 1)  (is ∀∞ n. ?P n)
  proof (elim eventually-rev-mp, intro eventually-sequentiallyI impI)
    fix n :: nat assume A: Suc k ≤ n 0 < p n ∧ p n < 1
    then interpret pG: edge-space n p n by unfold-locales auto
    have 1 ≤ n using A by auto

    define mean-short-count where mean-short-count = (∫ es. short-count (?ug n
es) ∂ pG.P)

    have mean-short-count-le: mean-short-count ≤ (k − 2) * n powr (ε * k)
    proof −
      have small-empty: ⋀es k. k ≤ 2 ⟹ short-cycles (edge-space.edge-ugraph n
es) k = {}
        by (auto simp add: short-cycles-def ucycles-def)
      have short-count-conv: ⋀es. short-count (?ug n es) = (∑ i=3..k. real (card
{c ∈ ucycles (?ug n es). uwalk-length c = i}))
      proof (unfold short-count-def, induct k)
        case 0 with small-empty show ?case by auto
      next
        case (Suc k)
        show ?case proof (cases Suc k ≤ 2)
          case True with small-empty show ?thesis by auto
        next
          case False
          have {c ∈ ucycles (?ug n es). uwalk-length c ≤ Suc k}
              = {c ∈ ucycles (?ug n es). uwalk-length c ≤ k} ∪ {c ∈ ucycles (?ug n
es). uwalk-length c = Suc k}
            by auto
          moreover
          have finite (uverts (edge-space.edge-ugraph n es)) by auto
          ultimately
          have card {c ∈ ucycles (?ug n es). uwalk-length c ≤ Suc k}
           = card {c ∈ ucycles (?ug n es). uwalk-length c ≤ k} + card {c ∈ ucycles
(?ug n es). uwalk-length c = Suc k}
            using finite-ucycles by (subst card-Un-disjoint[symmetric]) auto
          then show ?thesis
            using Suc False unfolding short-cycles-def by (auto simp: not-le)
        qed
      qed

      have mean-short-count = (∑ i=3..k. ∫ es. card {c ∈ ucycles (?ug n es).
```

```
uwalk-length c = i} ∂ pG.P)
      unfolding mean-short-count-def short-count-conv
    by (subst Bochner-Integration.integral-sum) (auto intro: pG.integral-finite-singleton)
    also have ... = (∑ i∈{3..k}. of-nat (fact n div fact (n − i)) * p n ^ i)
      using A by (simp add: pG.mean-k-cycles)
    also have ... ≤ (∑ i∈{3..k}. n ^ i * p n ^ i)
      apply (rule sum-mono)
        by (meson A fact-div-fact-le-pow Suc-leD atLeastAtMost-iff of-nat-le-iff
order-trans mult-le-cancel-right-pos zero-less-power)
    also have ... ≤ (∑ i∈{3..k}. n powr (ε * k))
      using ‹1 ≤ n› ‹0 < ε› A
     by (intro sum-mono) (auto simp: p-def field-simps powr-mult-base powr-powr
        powr-realpow[symmetric] powr-mult[symmetric] powr-add[symmetric])
    finally show ?thesis by simp
  qed

  have pG.prob {es ∈ space pG.P. n/2 ≤ short-count (?ug n es)} ≤ mean-short-count
/ (n/2)
    unfolding mean-short-count-def using ‹1 ≤ n›
    by (intro pG.Markov-inequality) (auto simp: short-count-def)
  also have ... ≤ 2 * (k − 2) * n powr (ε * k − 1)
  proof −
    have mean-short-count / (n / 2) ≤ 2 * (k − 2) * (1 / n powr 1) * n powr
(ε * k)
      using mean-short-count-le ‹1 ≤ n› by (simp add: field-simps)
    then show ?thesis by (simp add: powr-diff algebra-simps)
  qed
  finally show ?P n .
 qed

 define pf-short-count pf-α
  where pf-short-count n = probGn p n (λes. n/2 ≤ short-count (?ug n es))
   and pf-α n = probGn p n (λes. 1/2 * n/k ≤ α (edge-space.edge-ugraph n es))
  for n

 have ev-short-count-le: ∀∞ n. pf-short-count n < 1 / 2
 proof −
  have ε * k − 1 < 0
    using ε-props ‹3 ≤ k› by (auto simp: field-simps)
  then have (λn. 2 * (k − 2) * n powr (ε * k − 1)) ⟶ 0 (is ?bound ⟶
0)
    by (intro tendsto-mult-right-zero LIMSEQ-neg-powr)
  then have ∀∞ n. dist (?bound n) 0 < 1 / 2
    by (rule tendstoD) simp
  with prob-short-count-le show ?thesis
    by (rule eventually-elim2) (auto simp: dist-real-def pf-short-count-def)
 qed

 have lim-α: pf-α ⟶ 0
```

```
 proof −
  have 0 < k using ‹3 ≤ k› by simp

  have ∀∞ n. (6*k) * ln n / n ≤ p n ⟷ (6*k) * ln n * n powr − ε ≤ 1
  proof (rule eventually-sequentiallyI)
   fix n :: nat assume 1 ≤ n
    then have (6 * k) * ln n / n ≤ p n ⟷ (6*k) * ln n * (n powr − 1) ≤ n
powr (ε − 1)
      by  (subst powr-minus) (simp add: divide-inverse p-def)
     also have ... ⟷ (6*k) * ln n * ((n powr − 1) / (n powr (ε − 1))) ≤ n
powr (ε − 1) / (n powr (ε − 1))
      using ‹1 ≤ n› by (auto simp: field-simps)
    also have ... ⟷ (6*k) * ln n * n powr − ε ≤ 1
      by (simp add: powr-diff powr-minus divide-simps)
    finally show (6*k) * ln n / n ≤ p n ⟷ (6*k) * ln n * n powr − ε ≤ 1 .
  qed
  then have (∀∞ n. (6 * k) * ln n / real n ≤ p n)
     ⟷ (∀∞ n. (6*k) * ln n * n powr − ε ≤ 1)
    by (rule eventually-subst)
  also have ∀∞ n. (6*k) * ln n * n powr − ε ≤ 1
  proof −
    { fix n :: nat assume 0 < n
      have ln (real n) ≤ n powr (ε/2) / (ε/2)
        using ‹0 < n› ‹0 < ε› by (intro ln-powr-bound) auto
      also have ... ≤ 2/ε * n powr (ε/2) by (auto simp: field-simps)
      finally have (6*k) * ln n * (n powr − ε)  ≤ (6*k) * (2/ε * n powr (ε/2))
* (n powr − ε)
        using ‹0 < n› ‹0 < k› by (intro mult-right-mono mult-left-mono) auto
      also have ... = 12*k/ε * n powr (−ε/2)
        unfolding divide-inverse
        by (auto simp: field-simps powr-minus[symmetric] powr-add[symmetric])
      finally have (6*k) * ln n * (n powr − ε) ≤ 12*k/ε * n powr (−ε/2) .
    }
    then have ∀∞ n. (6*k) * ln n * (n powr − ε) ≤ 12*k/ε * n powr (−ε/2)
      by (intro eventually-sequentiallyI[of 1]) auto
    also have ∀∞ n. 12*k/ε * n powr (−ε/2) ≤ 1
    proof −
      have (λn. 12*k/ε * n powr (−ε/2)) ⟶ 0
        using ‹0 < ε› by (intro tendsto-mult-right-zero LIMSEQ-neg-powr) auto
      then show ?thesis
          using ‹0 < ε› by − (drule tendstoD[where e=1], auto elim: eventu-
ally-mono)
    qed
    finally (eventually-le-le) show ?thesis .
  qed
  finally have ∀∞ n. real (6 * k) * ln (real n) / real n ≤ p n .
 with ev-p ‹0 < k› show ?thesis unfolding pf-α-def by (rule almost-never-le-α)
 qed
```

from *ev-short-count-le lim-α*[*THEN tendstoD*, *of 1/2*] *ev-p*
have $\forall^\infty$ *n*. *0* < *p n* ∧ *p n* < *1* ∧ *pf-short-count n* < *1/2* ∧ *pf-α n* < *1/2*
by *simp* (*elim eventually-rev-mp*, *auto simp*: *eventually-sequentially dist-real-def*)
then obtain *n* **where** *0* < *p n p n* < *1* **and** [*arith*]: *0* < *n*
and *probs*: *pf-short-count n* < *1/2 pf-α n* < *1/2*
by (*auto simp*: *eventually-sequentially*)
then interpret *ES*: *edge-space n* (*p n*) **by** *unfold-locales auto*

have *rest-compl*: ⋀*A P*. *A* − {*x*∈*A*. *P x*} = {*x*∈*A*. ¬*P x*} **by** *blast*

from *probs* **have** *ES.prob* ({*es* ∈ *space ES.P*. *n/2* ≤ *short-count* (*?ug n es*)}
∪ {*es* ∈ *space ES.P*. *1/2* ∗ *n/k* ≤ *α* (*?ug n es*)}) ≤ *pf-short-count n* + *pf-α n*
unfolding *pf-short-count-def pf-α-def* **by** (*subst ES.finite-measure-subadditive*) *auto*
also have ... < *1* **using** *probs* **by** *auto*
finally have *0* < *ES.prob* (*space ES.P* − ({*es* ∈ *space ES.P*. *n/2* ≤ *short-count* (*?ug n es*)}
∪ {*es* ∈ *space ES.P*. *1/2* ∗ *n/k* ≤ *α* (*?ug n es*)})) (**is** *0* < *ES.prob ?S*)
by (*subst ES.prob-compl*) *auto*
also have *?S* = {*es* ∈ *space ES.P*. *short-count* (*?ug n es*) < *n/2* ∧ *α* (*?ug n es*) < *1/2*∗ *n/k*} (**is** ... = *?C*)
by (*auto simp*: *not-less rest-compl*)
finally have *?C* ≠ {} **by** (*intro notI*) (*simp only*:, *auto*)
then obtain *es* **where** *es-props*: *es* ∈ *space ES.P*
short-count (*?ug n es*) < *n/2* *α* (*?ug n es*) < *1/2* ∗ *n/k*
by *auto*
— now we obtained a high colored graph (few independent nodes) with almost no short cycles

define *G* **where** *G* = *?ug n es*
define *H* **where** *H* = *kill-short G k*

have *G-props*: *uverts G* = {*1..n*} *finite* (*uverts G*) *short-count G* < *n/2* *α G* < *1/2* ∗ *n/k*
unfolding *G-def* **using** *es-props* **by** (*auto simp*: *ES.S-verts-def*)

have *uwellformed G* **by** (*auto simp*: *G-def uwellformed-def all-edges-def ES.S-edges-def*)
with *G-props* **have** *T1*: *uwellformed H* **unfolding** *H-def* **by** (*intro kill-short-uwellformed*)

have *enat l* ≤ *enat k* **using** ‹*l* ≤ *k*› **by** *simp*
also have ... < *girth H* **using** *G-props* **by** (*auto simp*: *kill-short-large-girth H-def*)
finally have *T2*: *l* < *girth H* **.**

have *card-H*: *n/2* ≤ *card* (*uverts H*)
using *G-props es-props kill-short-order-of-graph*[*of G k*] **by** (*simp add*: *short-count-def H-def*)

then have *uverts-H*: *uverts H* ≠ {} *0* < *card* (*uverts H*) **by** *auto*

then have $0 < \alpha\ H$ **using** *zero-less-α uverts-H* **by** *auto*

have *α-HG*: $\alpha\ H \leq \alpha\ G$
unfolding *H-def G-def* **by** (*auto intro*: *kill-short-α*)

have $enat\ l \leq ereal\ k$ **using** ‹$l \leq k$› **by** *auto*
also have $\ldots < (n/2)\ /\ \alpha\ G$ **using** *G-props* ‹$3 \leq k$›
by (*cases* $\alpha\ G$) (*auto simp*: *field-simps*)
also have $\ldots \leq (n/2)\ /\ \alpha\ H$ **using** *α-HG* ‹$0 < \alpha\ H$›
by (*auto simp*: *ereal-of-enat-pushout intro*!: *ereal-divide-left-mono*)
also have $\ldots \leq card\ (uverts\ H)\ /\ \alpha\ H$ **using** *card-H* ‹$0 < \alpha\ H$›
by (*auto intro*!: *ereal-divide-right-mono*)
also have $\ldots \leq chromatic\text{-}number\ H$ **using** *uverts-H T1* **by** (*intro chromatic-lb*)
auto
finally have *T3*: $l < chromatic\text{-}number\ H$
by (*simp del*: *ereal-of-enat-simps*)

from *T1 T2 T3* **show** *?thesis* **by** *fast*
qed

end

References

[1] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, 4 edition, 2010. http://diestel-graph-theory.com.