Formalization of the Schwartz-Zippel Lemma

Sunpill Kim and Yong Kiam Tan

February 6, 2026

Abstract

This short entry formalizes a version of the Schwartz-Zippel lemma for probabilistic (multivariate) polynomial identity testing. The entry includes a textbook example using the lemma to test for perfect matchings in a bipartite graph. The lemma is attributed to several independent authors, including Schwartz [3], Zippel [4], and DeMillo and Lipton [1]; a historical perspective is given by Lipton [2].

Contents

1 The Schwartz-Zippel Lemma **1**

2 A Probabilistic Test for Perfect Matchings **12**

1 The Schwartz-Zippel Lemma

theory *Schwartz-Zippel* **imports**
Factor-Algebraic-Polynomial.Poly-Connection
Polynomials.MPoly-Type
HOL−Probability.Product-PMF
begin

This theory formalizes the Schwartz-Zippel lemma for multivariate polynomials (*mpoly*).

lemma *schwartz-zippel-uni*:
fixes P :: ($'a$::*idom*) *Polynomial.poly*
fixes S :: $'a$ *set*
assumes *finite* S $S \neq \{\}$
assumes *Polynomial.degree* $P \leq d$
assumes $P \neq 0$
shows *measure-pmf.prob* (*pmf-of-set* S) $\{r.$ *poly* P $r = 0\} \leq$ *real* d / *card* S
proof −
have *1*: $(\{r.$ *poly* P $r = 0\} \cap S) = \{r.$ *poly* P $r = 0 \wedge r \in S\}$
by *auto*
have *2*: *prob-space.prob* (*pmf-of-set* S) $\{r.$ *poly* P $r = 0\} =$
prob-space.prob (*pmf-of-set* S) $\{r.$ *poly* P $r = 0 \wedge r \in S\}$

```
   by (metis (full-types) 1 assms(1) assms(2) measure-Int-set-pmf set-pmf-of-set)
  have card: card {r. poly P r = 0  ∧ r ∈ S} ≤ d
     using poly-roots-degree assms(3) assms(4) by (smt (verit) Collect-mono-iff
card-mono le-trans poly-roots-finite)
  have inter: S ∩ {r. poly P r = 0 ∧ r ∈ S} =
    {r. poly P r = 0 ∧ r ∈ S}
    by auto

  have prob:prob-space.prob (pmf-of-set S) {r. poly P r = 0 ∧ r ∈ S} ≤ real d /
card S
    by (simp add: assms(1) assms(2) assms(4) card inter measure-pmf-of-set di-
vide-right-mono)

  thus prob-space.prob (pmf-of-set S) {r. poly P r = 0 } ≤ real d / card S
    by (simp add: 2)
qed



lemma degree-mpoly-to-poly [simp]:
  assumes vars p = {x}
  shows Polynomial.degree (mpoly-to-poly x p) = MPoly-Type.degree p x
proof (rule antisym)
  show Polynomial.degree (mpoly-to-poly x p) ≤ MPoly-Type.degree p x
  proof (intro Polynomial.degree-le allI impI)
    fix i assume i: i > MPoly-Type.degree p x
    have poly.coeff (mpoly-to-poly x p) i =
            (∑ m. 0 when lookup m x = i)
      unfolding coeff-mpoly-to-mpoly-poly using i
     by (metis (no-types, lifting) Sum-any.cong Sum-any.neutral coeff-mpoly-to-poly
degree-geI leD lookup-single-eq when-neq-zero)
    also have ... = 0
      by simp
    finally show poly.coeff (mpoly-to-poly x p) i = 0 .
  qed
next
  show Polynomial.degree (mpoly-to-poly x p) ≥ MPoly-Type.degree p x
  proof (cases p = 0)
    case False
   then obtain m where m: MPoly-Type.coeff p m ≠ 0 lookup m x = MPoly-Type.degree
p x
      using monom-of-degree-exists by blast
    show Polynomial.degree (mpoly-to-poly x p) ≥ MPoly-Type.degree p x
    proof (rule Polynomial.le-degree)
      have 0 ≠ MPoly-Type.coeff p m
        using m by auto
      have poly.coeff (mpoly-to-poly x p) (MPoly-Type.degree p x) =
        MPoly-Type.coeff p (monomial (MPoly-Type.degree p x) x)
        unfolding coeff-mpoly-to-poly by auto
```

also have *monomial* (*MPoly-Type.degree p x*) *x* = *m*
unfolding *m*(*2*)[*symmetric*] **using** *assms coeff-notin-vars m*(*1*)
by (*intro keys-subset-singleton-imp-monomial*) *blast*
finally show *poly.coeff* (*mpoly-to-poly x p*) (*MPoly-Type.degree p x*) ≠ *0*
using *m* **by** *auto*
qed
qed *auto*
qed

lemma *total-degree-add*: *total-degree* (*x* + *y*) ≤ *max* (*total-degree x*) (*total-degree y*)
proof −
define *h* :: (*nat* $\Rightarrow_0$ *nat*) ⇒ *nat* **where** *h* = (*λm. sum* (*lookup m*) (*keys m*))
have *insert 0* (*h* ' *keys* (*mapping-of* (*x* + *y*))) ⊆
insert 0 (*h* ' (*keys* (*mapping-of x*) ∪ *keys* (*mapping-of y*)))
unfolding *plus-mpoly.rep-eq*
by (*intro Set.insert-mono image-mono Poly-Mapping.keys-add*)
also have . . . = *insert 0* (*h* ' *keys* (*mapping-of x*)) ∪
insert 0 (*h* ' *keys* (*mapping-of y*))
by (*simp add*: *image-Un*)
finally have *Max* (*insert 0* (*h* ' *keys* (*mapping-of* (*x* + *y*)))) ≤ *Max* . . .
by (*intro Max-mono*) *simp-all*
also have *?this* ⟷ *?thesis*
unfolding *total-degree-def map-fun-def o-def id-def h-def* [*symmetric*]
by (*subst Max-Un* [*symmetric*]; *simp*)
finally show *?thesis* **.**
qed

lemma *total-degree-sum*:
assumes *finite S*
shows*total-degree* (*sum f S*) ≤
Max ((*total-degree* ∘ *f*) ' *S*)
using *assms*
proof (*induction S*)
case *empty*
then show *?case*
by *auto*
next
case (*insert x F*)
show *?case*
proof (*cases F* = {})
case *False*
have *total-degree* (*f x* + *sum f F*) ≤
max (*total-degree* (*f x*)) (*total-degree* (*sum f F*))
using *total-degree-add* **by** *blast*
moreover have ... ≤
max (*total-degree* (*f x*)) (*Max* ((*total-degree* ∘ *f*) ' *F*))

using *local.insert*(*3*) **by** *linarith*
also have ... = (*Max* ((*total-degree* ∘ *f*) ‘ *insert x F*))
using *insert False* **by** *simp*
finally show *?thesis*
using *insert.hyps* **by** *simp*
qed *simp-all*
qed

lemma *coeff-mpoly-to-mpoly-poly-restrict*:
shows *coeff* (*mpoly-to-mpoly-poly x P*) *i* =
sum (λ*m*.
MPoly-Type.monom (*remove-key x m*)
(*MPoly-Type.coeff P m*) *when lookup m x* = *i*)
(*Poly-Mapping.keys* (*mapping-of P*))
unfolding *coeff-mpoly-to-mpoly-poly*
by (*auto intro*!: *Sum-any.expand-superset simp*: *coeff-keys*)

lemma *Max-le-Max*:
assumes *A* ≠ {}
assumes *finite A finite B*
assumes ⋀*a*. *a* ∈ *A* ⟹ ∃ *b*. *b* ∈ *B* ∧ *a* ≤ *b*
shows *Max A* ≤ *Max B*
proof −
from *assms* **have** *B* ≠ {}
by *blast*
thus *?thesis*
using *assms* **by** (*intro Max.boundedI*) (*auto simp*: *Max-ge-iff*)
qed

lemma *sum-lookup-remove-key*:
sum (*lookup* (*remove-key x y*)) (*keys* (*remove-key x y*)) + *lookup y x* =
sum (*lookup y*) (*keys y*)
proof −
have *sum* (*lookup y*) (*keys y*) =
sum (*lookup y*) (*keys* (*remove-key x y* + *monomial* (*lookup y x*) *x*))
by (*subst* (*2*) *remove-key-sum* [*of x*, *symmetric*]) *auto*
also have *keys* (*remove-key x y* + *monomial* (*lookup y x*) *x*) =
keys (*remove-key x y*) ∪ *keys* (*monomial* (*lookup y x*) *x*)
by (*subst keys-add*) (*auto simp flip*: *remove-key-keys*)
also have *sum* (*lookup y*) ... = *sum* (*lookup y*) (*keys* (*remove-key x y*)) +
sum (*lookup y*) (*keys* (*monomial* (*lookup y x*) *x*))
by (*subst sum.union-disjoint*) (*auto simp flip*: *remove-key-keys*)
also have *sum* (*lookup y*) (*keys* (*remove-key x y*)) =
sum (*lookup* (*remove-key x y*)) (*keys* (*remove-key x y*))
by (*intro sum.cong*) (*auto simp*: *remove-key-lookup when-def simp flip*: *remove-key-keys*)
also have *sum* (*lookup y*) (*keys* (*monomial* (*lookup y x*) *x*)) =
sum (*lookup y*) {*x*}

```
    by (intro sum.mono-neutral-cong-left) auto
  also have ... = lookup y x
    by simp
  finally show ?thesis ..
qed

lemma total-degree-nonzero:
  assumes P ≠ 0
  shows total-degree P =
    Max ((λx. sum (lookup x) (keys x)) ‘ keys (mapping-of P))
  unfolding total-degree-def by (auto simp: assms)

lemma poly-mapping-eq-iff: (m = m′) ⟷ (∀ i. lookup m i = lookup m′ i)
  by (auto intro: poly-mapping-eqI)


lemma total-degree-coeff-mpoly-to-mpoly-poly:
  assumes coeff (mpoly-to-mpoly-poly x P) i ≠ 0
  shows total-degree (coeff (mpoly-to-mpoly-poly x P) i) + i ≤ total-degree P
proof −
  have P: P ≠ 0
    using assms by force

  have nonempty: keys
   (mapping-of(poly.coeff (mpoly-to-mpoly-poly x P) i)) ≠ {}
    using assms by auto

  have ∗:
    {y. ∃ xa∈keys (mapping-of P) ∩ {xa. lookup xa x = i} ∩ {x. MPoly-Type.coeff
P x ≠ 0}. y = sum (lookup xa) (keys xa)} ⊆
    (insert 0
        ((λy. sum (lookup y) (keys y)) ‘ keys (mapping-of P)))
  by auto

  have total-degree (coeff (mpoly-to-mpoly-poly x P) i) + i =
    (MAX x∈keys (mapping-of (poly.coeff (mpoly-to-mpoly-poly x P) i)).
      sum (lookup x) (keys x)) + i
    by (subst total-degree-nonzero) (use assms in auto)
  moreover have ... =
      (MAX x∈keys (mapping-of (poly.coeff (mpoly-to-mpoly-poly x P) i)).
      sum (lookup x) (keys x) + i)
    by (subst Max-add-commute[symmetric]) (use assms in auto)
  moreover have ... =
    (MAX x∈⋃ xa∈keys (mapping-of P) ∩
             {xa. lookup xa x = i} ∩
             {x. MPoly-Type.coeff P x ≠ 0}.
            {remove-key x xa}.
      sum (lookup x) (keys x) + i)
    unfolding coeff-mpoly-to-mpoly-poly-restrict
```

unfolding *MPoly-Type.degree-def mapping-of-sum*[*symmetric*]
apply (*subst keys-sum*)
subgoal by *simp*
subgoal for *i j*
apply (*simp add*: *zero-mpoly.rep-eq poly-mapping-eq-iff remove-key-lookup when-def*)
apply *metis*
done
subgoal
by (*auto simp add*: *if-distrib zero-mpoly.rep-eq when-def*)
done
moreover have ... =
Max {*y*. ∃ *xa*∈*keys* (*mapping-of P*) ∩ {*xa*. *lookup xa x* = *i*} ∩
{*x*. *MPoly-Type.coeff P x* ≠ *0*}.
y = *sum* (*lookup* (*remove-key x xa*)) (*keys* (*remove-key x xa*)) + *lookup xa x*}
by (*intro arg-cong*[**where** *f* = *Max*]) *auto*
moreover have ... =
Max {*y*. ∃ *xa*∈*keys* (*mapping-of P*) ∩ {*xa*. *lookup xa x* = *i*} ∩
{*x*. *MPoly-Type.coeff P x* ≠ *0*}.
y = *sum* (*lookup xa*) (*keys xa*)}
by (*subst sum-lookup-remove-key*) *auto*

moreover have ... ≤
Max (*insert 0*
((λ*y*. *sum* (*lookup y*) (*keys y*)) ‘ *keys* (*mapping-of P*)))
apply (*intro Max-mono*[*OF* ∗])
subgoal
using *nonempty* **unfolding** *coeff-mpoly-to-mpoly-poly-restrict*
by (*force elim*!: *sum.not-neutral-contains-not-neutral*)
subgoal
by *simp*
done
moreover have ... = *total-degree P*
unfolding *total-degree-def* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *degree-le-total-degree*:
shows *MPoly-Type.degree P x* ≤ *total-degree P*
unfolding *total-degree-def degree.rep-eq map-fun-def o-def id-def*
proof (*rule Max.boundedI*; *safe?*)
fix *k*
assume *k*: *k* ∈ *keys* (*mapping-of P*)
have *x* ∈ *keys k* ∨ *x* ∉ *keys k* **by** *auto*
moreover {
assume *x* ∈ *keys k*
then have *lookup k x* ≤ *sum* (*lookup k*) (*keys k*)
by (*simp add*: *sum.remove*)

then have *lookup k x*
≤ *Max* (*insert 0* ((λ*x*. *sum* (*lookup x*) (*keys x*)) ‘ *keys* (*mapping-of P*)))
using *k*
by (*smt* (*verit*, *ccfv-SIG*) *Max-ge dual-order.trans finite-imageI finite-insert finite-keys image-subset-iff subset-insertI*)
}
moreover {
assume *x* ∉ *keys k*
then have *lookup k x* = *0*
by (*simp add*: *in-keys-iff*)
then have *lookup k x*
≤ *Max* (*insert 0* ((λ*x*. *sum* (*lookup x*) (*keys x*)) ‘ *keys* (*mapping-of P*)))
by *linarith*
}
ultimately show *lookup k x*
≤ *Max* (*insert 0* ((λ*x*. *sum* (*lookup x*) (*keys x*)) ‘ *keys* (*mapping-of P*))) **by** *auto*
qed *auto*

lemma *insertion-update*:
shows *insertion* (*f*(*x* := *r*)) *P* = *poly* (*map-poly* (*insertion f*) (*mpoly-to-mpoly-poly x P*)) *r*
by (*subst insertion-mpoly-to-mpoly-poly*[*symmetric*,**where** β = *f*]) *auto*

lemma *measure-pmf-prob-dependent-product-bound*:
assumes *countable A* ⋀*i*. *countable* (*B i*)
assumes ⋀*a*. *a* ∈ *A* ⟹ *measure-pmf.prob N* (*B a*) ≤ *r*
shows *measure-pmf.prob* (*pair-pmf M N*) (*Sigma A B*) ≤ *measure-pmf.prob M A* ∗ *r*
proof −
have *measure-pmf.prob* (*pair-pmf M N*) (*Sigma A B*) = (∑$_a$(*a*, *b*)∈*Sigma A B*. *pmf M a* ∗ *pmf N b*)
by (*auto intro*!: *infsetsum-cong simp add*: *measure-pmf-conv-infsetsum pmf-pair*)
moreover have . . . = (∑$_a$*a*∈*A*. ∑$_a$*b*∈*B a*. *pmf M a* ∗ *pmf N b*)
proof (*subst infsetsum-Sigma*)
show (λ(*a*, *b*). *pmf M a* ∗ *pmf N b*) *abs-summable-on Sigma A B*
unfolding *case-prod-unfold*
by (*metis* (*no-types*, *lifting*) *SigmaE abs-summable-on-cong pmf-abs-summable pmf-pair prod.sel*)
qed (*use assms* **in** *auto*)
moreover have ... = (∑$_a$*a*∈*A*. *pmf M a* ∗ (*measure-pmf.prob N* (*B a*)))
by (*simp add*: *infsetsum-cmult-right measure-pmf-conv-infsetsum pmf-abs-summable*)
moreover have ... ≤ (∑$_a$*a*∈*A*. *pmf M a* ∗ *r*)
proof (*rule infsetsum-mono*)
show (λ*a*. *pmf M a* ∗ *measure-pmf.prob N* (*B a*)) *abs-summable-on A*
proof (*rule abs-summable-on-comparison-test′*)
show *norm* (*pmf M x* ∗ *measure-pmf.prob N* (*B x*)) ≤ *pmf M x* ∗ *1* **for** *x*

unfolding *norm-mult* **by** (*intro mult-mono*) *auto*
qed *auto*
qed (*auto intro*: *mult-mono assms*)
moreover have... = $r * (\sum_a a{\in}A.\ pmf\ M\ a)$
by (*simp add*: *infsetsum-cmult-left pmf-abs-summable*)
moreover have... = *measure-pmf.prob M A* ∗ *r*
by (*simp add*: *measure-pmf-conv-infsetsum*)
ultimately show *?thesis*
by *linarith*
qed

lemma *measure-pmf-prob-dependent-product-bound′*:
assumes *countable* (*A* ∩ *set-pmf M*) ⋀*i*. *countable* (*B i* ∩ *set-pmf N*)
assumes ⋀*a*. *a* ∈ *A* ∩ *set-pmf M* ⟹ *measure-pmf.prob N* (*B a* ∩ *set-pmf N*) ≤ *r*
shows *measure-pmf.prob* (*pair-pmf M N*) (*Sigma A B*) ≤ *measure-pmf.prob M A* ∗ *r*
proof −
have ∗: *Sigma A B* ∩ (*set-pmf M* × *set-pmf N*) =
Sigma (*A* ∩ *set-pmf M*) (λ*i*. *B i* ∩ *set-pmf N*)
by *auto*

have *measure-pmf.prob* (*pair-pmf M N*) (*Sigma A B*) =
measure-pmf.prob (*pair-pmf M N*) (*Sigma* (*A* ∩ *set-pmf M*) (λ*i*. *B i* ∩ *set-pmf N*))
by (*metis* ∗ *measure-Int-set-pmf set-pair-pmf*)
moreover have ... ≤
measure-pmf.prob M (*A* ∩ *set-pmf M*) ∗ *r*
using *measure-pmf-prob-dependent-product-bound*[*OF assms*(*1*−*3*)]
by *auto*
moreover have ... = *measure-pmf.prob M A* ∗ *r*
by (*simp add*: *measure-Int-set-pmf*)
ultimately show *?thesis* **by** *linarith*
qed

lemma *finite-set-pmf-Pi-pmf*:
assumes *finite A*
assumes ⋀*x*. *x* ∈ *A* ⟹ *finite* (*set-pmf* (*p x*))
shows *finite* (*set-pmf* (*Pi-pmf A def p*))
proof −
from *set-Pi-pmf-subset′*[*OF assms*(*1*)]
have *1*: *set-pmf* (*Pi-pmf A def p*)
⊆ *PiE-dflt A def* (*set-pmf* ∘ *p*)
by *fastforce*
have *2*: *finite* ...
by (*intro finite-PiE-dflt*) (*use assms* **in** *auto*)
show *?thesis* **using** *1 2*

using *finite-subset* **by** *auto*
qed

theorem *schwartz-zippel*:
fixes P :: ('*a*::*idom*) *mpoly*
fixes S :: '*a set*
assumes S: *finite* S $S \neq \{\}$
assumes V: *finite* V
assumes P: *total-degree* $P \leq d$ $P \neq 0$ *vars* $P \subseteq V$
shows *measure-pmf.prob* (*Pi-pmf* V *0* (λi. *pmf-of-set* S)) $\{f.$ *insertion f* $P = 0\}$ $\leq$ *real d* / *card S*
using V P
proof (*induction V arbitrary*: *P d*)
case *empty*
then show *?case*
by (*metis* (*mono-tags*, *lifting*) *Collect-empty-eq divide-nonneg-nonneg insertion-Const lead-coeff-eq-0-iff measure-empty of-nat-0-le-iff subset-empty vars-empty-iff*)
next
case (*insert x V*)
have $x \notin$ *vars* $P \vee x \in$ *vars* P **by** *auto*
moreover **{**
assume x: $x \notin$ *vars* P
have $*$: *prob-space.prob* (*Pi-pmf* V *0* (λi. *pmf-of-set* S)) $\{f.$ *insertion f* $P = 0\}$ $\leq$ *real d* / *card S*
by (*smt* (*verit*) *Collect-cong Diff-insert2 Diff-insert-absorb* ‹$x \notin$ *vars* P› *diff-shunt-var insert.IH insert.prems*(*3*) *insert-Diff-single local.insert*(*2*) *local.insert*(*4*) *local.insert*(*5*) *subset-insertI*)

have *inser*: *Pi-pmf* V *0* (λi. *pmf-of-set* S) =
map-pmf (λf. $f(x := 0)$) ((*Pi-pmf* (*insert x V*) *0* (λi. *pmf-of-set* S)))
by (*metis Diff-insert-absorb Pi-pmf-remove finite-insert local.insert*(*1*) *local.insert*(*2*) *local.insert*(*6*))

have *f-aux*: *insertion* ($f(x := 0)$) $P = 0 \longleftrightarrow$ *insertion f* $P = 0$ **for** *f*
by (*metis x array-rules*(*4*) *insertion-irrelevant-vars*)
have *f*: (λf. $f(x := 0)$)−‘$\{f.$ *insertion f* $P = 0 \wedge f\ x = 0\}$ = $\{f.$ *insertion f* $P = 0\}$
by (*simp add*: *f-aux*)

have *prob-space.prob* ((*Pi-pmf* (*insert x V*) *0* (λi. *pmf-of-set* S)))
$\{f.$ *insertion f* $P = 0\}$ = *prob-space.prob* (*Pi-pmf* V *0* (λi. *pmf-of-set* S)) $\{f.$ *insertion f* $P = 0$ $\}$
using *inser f* **by** *fastforce*
moreover have ... $\leq$ *real d* / *card S*
by (*simp add*: $*$)
ultimately have *?case*
by *linarith*
}
moreover **{**

assume *x*: $x \in$ *vars P*
define *px* **where** *px* = *mpoly-to-mpoly-poly x P*
define *q* **where** *q* = *Polynomial.lead-coeff px*

have *xnV*: $x \notin V$
  by (*simp add*: *local.insert*(*2*))

have *split*: {*f*. *insertion f P* = *0*} ⊆
  {*f*. *insertion f q* = *0*} ∪
  {*f*. *insertion f q* ≠ *0* ∧ *insertion f P* = *0*} (**is** - ⊆ *?E2* ∪ *?E12*) **by** *blast*

obtain *l* **where** *l*: *MPoly-Type.degree P x* = *l* **by** *simp*

have *ld*: $l \leq d$
  using *l degree-le-total-degree le-trans local.insert*(*4*) **by** *blast*

have *varq*: *vars q* ⊆ *V*
  by (*metis x dual-order.trans insert.prems*(*3*) *px-def q-def subset-insert-iff vars-coeff-mpoly-to-mpoly-poly*)

have *dl*: *degree* (*mpoly-to-mpoly-poly x P*) = *l*
  by (*simp add*: *l*)
have *qnz*: *q* ≠ *0*
  unfolding *q-def*
  by (*metis degree-0 degree-eq-0-iff dl l leading-coeff-neq-0 px-def x*)

have *total-degree P* ≥
  degree (*mpoly-to-mpoly-poly x P*) +
  total-degree (*Polynomial.lead-coeff* (*mpoly-to-mpoly-poly x P*))
  by (*metis Groups.add-ac*(*2*) *px-def q-def qnz total-degree-coeff-mpoly-to-mpoly-poly*)

then have *tdq*: *total-degree q* ≤ *d*−*l*
  using *local.insert*(*4*) *px-def q-def dl* **by** *force*

from *insert.IH*[*OF tdq qnz varq*]
have ∗: *measure-pmf.prob* (*Pi-pmf V 0* (λ*i*. *pmf-of-set S*)) {*f*. *insertion f q* = *0*} ≤ *real* (*d*−*l*) /*real* (*card S*) **by** *auto*

have *inser*: *Pi-pmf V 0* (λ*i*. *pmf-of-set S*) =
  map-pmf (λ*f*. *f*(*x* := *0*)) ((*Pi-pmf* (*insert x V*) *0* (λ*i*. *pmf-of-set S*)))
  by (*metis Diff-insert-absorb Pi-pmf-remove finite-insert local.insert*(*1*) *local.insert*(*2*) *local.insert*(*6*))

have *aux*: *insertion* (*f*(*x* := *0*)) *q* = *0* ⟷ *insertion f q* = *0* **for** *f*
  by (*metis* ‹*vars q* ⊆ *V*› *array-rules*(*4*) *insertion-irrelevant-vars local.insert*(*2*) *subsetD*)
have (λ*f*. *f*(*x* := *0*))−‘{*f*. *insertion f q* = *0* ∧ *f x* = *0*} = {*f*. *insertion f q* =

0}
  by (*simp add*: *aux*)

  then have *prob-space.prob* (*Pi-pmf* (*insert x V*) *0* (λ*i*. *pmf-of-set S*)) {*f*. *insertion f q* = *0*} =
    prob-space.prob (*map-pmf* (λ*f*. *f*(*x* := *0*)) (*Pi-pmf* (*insert x V*) *0* (λ*i*. *pmf-of-set S*))) {*f*. *insertion f q* = *0* ∧ *f x* = *0*}
  using *local.insert*(*6*) **by** *force*
  moreover have ... = *prob-space.prob* (*Pi-pmf V 0* (λ*i*. *pmf-of-set S*)) {*f*. *insertion f q* = *0* }
  using *inser* **by** *fastforce*

 ultimately have *e2*: *measure-pmf.prob* (*Pi-pmf* (*insert x V*) *0* (λ*i*. *pmf-of-set S*)) *?E2* ≤ (*d* − *l*) / *card S*
  using ∗ **by** *presburger*

 have *ib*: *prob-space.prob* (*pmf-of-set S*) {*r*. *poly* (*map-poly* (*insertion f*) *px*) *r* = *0*}
   ≤ *real l* / *real* (*card S*) **if** *insertion f q* ≠ *0* **for** *f*
 proof (*rule schwartz-zippel-uni*[*OF S*])
  show *Polynomial.degree* (*map-poly* (*insertion f*) *px*) ≤ *l*
   unfolding *px-def* **using** *that dl degree-map-poly-le* **by** *blast*
  show *map-poly* (*insertion f*) *px* ≠ *0*
   by (*metis Ring-Hom-Poly.degree-map-poly degree-0 degree-eq-0-iff dl l px-def q-def that x*)
 qed

 have *insertion-upd*: *insertion* (*f*(*x* := *r*)) *q* = *insertion f q* **for** *f r*
  by (*metis array-rules*(*4*) *insertion-irrelevant-vars local.insert*(*2*) *subsetD varq*)

 then have ∗: {*y*. *insertion* ((*fst y*)(*x* := *snd y*)) *q* ≠ *0* ∧
   insertion ((*fst y*)(*x* := *snd y*)) *P* = *0*} =
  Sigma {*f*. *insertion f q* ≠ *0*} (λ*f*. {*r*. *insertion* (*f*(*x* := *r*)) *P* = *0*})
  by *auto*

 have *measure-pmf.prob* (*Pi-pmf* (*insert x V*) *0* (λ*i*. *pmf-of-set S*)) *?E12* =
   measure-pmf.prob (*pair-pmf* (*Pi-pmf V 0* (λ*i*. *pmf-of-set S*)) (*pmf-of-set S*))
    ((λ(*f*, *y*). *f*(*x* := *y*)) −‘ {*f*. *insertion f q* ≠ *0* ∧ *insertion f P* = *0*})
  by (*subst Pi-pmf-insert*)
   (*use xnV insert* **in** ‹*auto simp add*: *pair-commute-pmf*[*of pmf-of-set S*]
       case-prod-unfold ∗ *px-def insertion-update*[*symmetric*]›)
 also have (λ(*f*, *y*). *f*(*x* := *y*)) −‘ {*f*. *insertion f q* ≠ *0* ∧ *insertion f P* = *0*} =
    Sigma {*f*. *insertion f q* ≠ *0*} (λ*f*. {*r*. *poly* (*map-poly* (*insertion f*) *px*) *r* = *0*})
  by (*auto simp*: *Sigma-def case-prod-unfold insertion-upd px-def simp flip*: *insertion-update*)

 also have *measure-pmf.prob* (*pair-pmf* (*Pi-pmf V 0* (λ*i*. *pmf-of-set S*)) (*pmf-of-set*

S)) ... ≤
  measure-pmf.prob (*Pi-pmf V 0* (λ*i. pmf-of-set S*)) {*f. insertion f q* ≠ *0*} ∗
  (*real l* / *real* (*card S*))
  by (*intro measure-pmf-prob-dependent-product-bound′*) (*auto simp*: *ib measure-Int-set-pmf*)
  also have ... ≤ *real l* / *real* (*card S*)
  by (*intro mult-left-le-one-le*) *auto*
  finally have *e12*: *measure-pmf.prob* (*Pi-pmf* (*insert x V*) *0* (λ*i. pmf-of-set S*)) *?E12* ≤ *l* / *card S* **.**

  have *measure-pmf.prob* (*Pi-pmf* (*insert x V*) *0* (λ*i. pmf-of-set S*)) {*f. insertion f P* = *0*} ≤
  measure-pmf.prob (*Pi-pmf* (*insert x V*) *0* (λ*i. pmf-of-set S*)) ({*f. insertion f q* = *0*} ∪ {*f. insertion f q* ≠ *0* ∧ *insertion f P* = *0*})
  by (*intro measure-pmf.finite-measure-mono*) (*use split* **in** *auto*)

  moreover have ... ≤
  measure-pmf.prob (*Pi-pmf* (*insert x V*) *0* (λ*i. pmf-of-set S*)) *?E2* +
  measure-pmf.prob (*Pi-pmf* (*insert x V*) *0* (λ*i. pmf-of-set S*)) *?E12*
  by (*simp add*: *measure-Un-le*)
  moreover have ... ≤ (*d*−*l*) / *card S* + *l* / *card S*
  using *e2 e12* **by** *auto*

  moreover have ... ≤ *real d* / *card S*
  by (*simp add*: *ld diff-divide-distrib of-nat-diff*)

  ultimately have *?case*
  by *linarith*
  }
  ultimately show *?case* **by** *auto*
qed

end

2 A Probabilistic Test for Perfect Matchings

theory *Rand-Perfect-Matching* **imports**
  Schwartz-Zippel
  Jordan-Normal-Form.Determinant
begin

We use a simple representation of bipartite graphs (with same no. vertices) *V*::*nat*, *E*::(*nat* × *nat*) *list* where *V* is the number of vertices in each partition and (*x*,*y*) ∈ *E* represents an edge between vertex *x* in the left partition and vertex *y* in the right partition.

definition *is-matching*::
  (*nat* × *nat*) *set* ⇒ (*nat* × *nat*) *set* ⇒ *bool*
  where *is-matching E match* ⟷

```
    match ⊆ E ∧
    inj-on fst match ∧
    inj-on snd match

definition has-perfect-matching::
  nat ⇒ (nat × nat) set ⇒ bool
  where has-perfect-matching V E ⟷
  (∃ match. is-matching E match ∧ card match = V)


definition adj-mat::nat ⇒ (nat × nat) set ⇒
    int mpoly mat
  where adj-mat V E =
    mat V V (λ(i,j).
      if (i,j) ∈ E then Var (i∗V+j) else 0)

lemma adj-mat-square[simp]:
  shows
    dim-row (adj-mat V E) = V
    dim-col (adj-mat V E) = V
  unfolding adj-mat-def
  by auto

lemma perfect-match-set-map-fst:
  assumes E ⊆ {0..<V} × {0..<V}
  assumes is-matching E match
  assumes card match = V
  shows   fst ‘ match = {0..<V}
proof −
  have fst ‘ match ⊆ fst ‘ E
    using assms(2) unfolding is-matching-def
    by auto
  moreover have ... ⊆ {0..<V}
    using assms(1) by auto
  ultimately have 1: fst‘ match ⊆ {0..<V}
    by auto
  then have 2: {0..<V} ⊆ fst ‘ match
    by (metis assms(2) assms(3) card-atLeastLessThan card-image card-subset-eq
diff-zero finite-atLeastLessThan is-matching-def)
  show ?thesis using 1 2 by auto
qed

lemma perfect-match-set-map-snd:
  assumes E ⊆ {0..<V} × {0..<V}
  assumes is-matching E match
  assumes card match = V
  shows snd ‘ match = {0..<V}
proof −
  have snd ‘ match ⊆ snd ‘ E
```

```
    using assms(2) unfolding is-matching-def
    by auto
  moreover have ... ⊆ {0..<V}
    using assms(1) by auto
  ultimately have 1: snd ‘ match ⊆ {0..<V}
    by auto
  then have 2: {0..<V} ⊆ snd ‘ match
     by (metis assms(2) assms(3) card-atLeastLessThan card-image card-subset-eq
diff-zero finite-atLeastLessThan is-matching-def)
  show ?thesis using 1 2 by auto
qed

lemma is-matching-permutes:
  assumes E ⊆ {0..<V} × {0..<V}
  assumes is-matching E match
  assumes card match = V
  obtains f where
    f permutes {0..<V}
    ⋀i. i < V ⟹ (i, f i) ∈ E
proof −
  have fV: fst ‘ match  = {0..<V}
    using perfect-match-set-map-fst[OF assms(1−3)] .

  have sV: snd ‘ match = {0..<V}
    using perfect-match-set-map-snd[OF assms(1−3)] .

  define f where f =
    (λi. if i < V then (@j. j < V ∧ (i,j) ∈ match) else i)

  have exuniq: i < V ⟹
    (∃!j. j < V ∧ (i,j) ∈ match) for i
  proof −
    assume i < V
    then have i ∈ fst ‘ match using fV
      by auto
    then obtain j where ij: (i,j) ∈ match by auto
    then have j ∈ snd ‘ match
      by (auto simp add: rev-image-eqI)
    then have 1: j < V using sV by auto

    thus ?thesis
      using ij 1
      apply auto[1]
      by (metis Pair-inject assms(2) fst-eqD inj-on-def is-matching-def)
  qed

  have 1: inj-on f {0..<V}
    unfolding f-def inj-on-def
    apply clarsimp
```

apply (*drule exuniq*)
apply (*drule exuniq*)
by (*smt* (*verit*) *assms*(*2*) *inj-on-def is-matching-def prod.sel*(*2*) *prod.simps*(*1*) *verit-sko-ex*)

have *2*: $f \in \{0..<V\} \rightarrow \{0..<V\}$
unfolding *f-def*
apply *clarsimp*
apply (*drule exuniq*)
by (*smt* (*verit*, *ccfv-threshold*) *verit-sko-ex*)

have *1*: *f permutes* $\{0..<V\}$
by (*intro inj-on-nat-permutes*[*OF 1 2*]) (*auto simp*: *f-def*)
have *2*: $i < V \Longrightarrow (i, f\ i) \in E$ **for** *i*
unfolding *f-def*
apply *clarsimp*
apply (*drule exuniq*)
using *assms*(*2*) **unfolding** *is-matching-def subset-iff*
by (*smt* (*verit*, *ccfv-threshold*) *verit-sko-ex*)

show *?thesis* **using** *that 1 2* **by** *auto*
qed

lemma *Var-not-0*:
shows $Var\ x \neq (0::'a::idom\ mpoly)$
by (*simp add*: *Var-altdef*)

lemma *sum-monom-0-iff*:
assumes *fin*: *finite S*
and *g*: $\bigwedge i\ j.\ i \in S \Longrightarrow j \in S \Longrightarrow g\ i = g\ j \Longrightarrow i = j$
shows $sum\ (\lambda\ i.\ MPoly\text{-}Type.monom\ (g\ i)\ (f\ i))\ S = 0 \longleftrightarrow (\forall\ i \in S.\ f\ i = 0)$ (**is** *?l* = *?r*)
proof −
{
assume ¬ *?r*
then obtain *i* **where** *i*: $i \in S$ **and** *fi*: $f\ i \neq 0$ **by** *auto*
let $?g = \lambda\ i.\ MPoly\text{-}Type.monom\ (g\ i)\ (f\ i)$
have $MPoly\text{-}Type.coeff\ (sum\ ?g\ S)\ (g\ i) =$
$f\ i + sum\ (\lambda\ j.\ MPoly\text{-}Type.coeff\ (?g\ j)\ (g\ i))\ (S - \{i\})$
by (*simp add*: *More-MPoly-Type.coeff-monom sum.remove*[*OF fin i*])
also have $sum\ (\lambda\ j.\ MPoly\text{-}Type.coeff\ (?g\ j)\ (g\ i))\ (S - \{i\}) = 0$
by (*smt* (*verit*, *ccfv-threshold*) *DiffE More-MPoly-Type.coeff-monom assms*(*2*) *i insert-iff sum.neutral when-simps*(*2*))
finally have $MPoly\text{-}Type.coeff\ (sum\ ?g\ S)\ (g\ i) \neq 0$ **using** *fi* **by** *auto*
hence ¬ *?l*
by *fastforce*
}
thus *?thesis* **by** *auto*

qed

lemma *prod-Var*:
assumes*finite S*
shows *prod* (*λi. Var*(*f i*)) *S* =
MPoly-Type.monom (*sum* (*λi. monomial 1* (*f i*)) *S*) *1*
using *assms*
proof (*induction S*)
case *empty*
then show *?case*
by *auto*
next
case (*insert x F*)
then show *?case*
by (*auto simp add*: *Var-altdef MPoly-Type.mult-monom*)
qed

lemma *det-adj-mat*:
shows *det* (*adj-mat V E*) =
($\sum$ *p* | *p permutes* {*0*..<*V*}.
MPoly-Type.monom (
sum (*λi.*
monomial 1 (*i* ∗ *V* + *p i*)) {*0*..<*V*})
(*if* ∀ *i* < *V*. (*i*,*p i*) ∈ *E then*
of-int (*sign p*)
else 0))
unfolding *det-def*
by (*force intro*!: *sum.cong simp add*: *adj-mat-def prod-Var monom-uminus sign-def*)

lemma *vars-prod-Var*:
assumes *finite S*
shows*vars* (*prod Var S*) = *S*
apply (*subst prod-Var*[*OF assms*])
apply (*subst vars-monom-keys*)
subgoal
by *simp*
apply (*subst keys-sum*[*OF assms*])
by *auto*

lemma *prod-Var-eq*:
assumes *finite S finite T*
assumes *prod Var S* = *prod Var T*
shows *S* = *T*
using *assms vars-prod-Var*
by *metis*

lemma *pair-enc-eq*:
assumes *a* ∗ *V* + *b* = *c* ∗ *V* + *d*
assumes *b* < *V d* < *V*

```
  shows b = (d::nat)
  by (metis div-eq-0-iff add-right-cancel assms(1) assms(2) assms(3) diff-add-inverse
div-mult-self3 mult-eq-0-iff)

lemma sum-monomial-eq:
  assumesf permutes {0..<V}
  assumesg permutes {0..<V}
  assumes
    (∑ i = 0..<V.
      monomial (1::nat) (i * V + (f i::nat))) =
    (∑ i = 0..<V.
      monomial (1::nat) (i * V + g i))
  shows f = g
proof −
  have injf: inj-on (λi. i * V + f i) {0..<V} and injg:inj-on (λi. i * V + g i)
{0..<V}
    unfolding inj-on-def
    using assms
    by (metis permutes-nat-less pair-enc-eq atLeastLessThan-iff permutes-def)+

  have MPoly-Type.monom (sum (λi. monomial (1::nat) (i * V + f i)) {0..<V})
1 =
        MPoly-Type.monom (sum (λi. monomial (1::nat)(i * V + g i)) {0..<V})
1
    using assms(3) by auto

  then have prod (λi. Var(i * V + f i)) {0..<V} =
    prod (λi. Var(i * V + g i)) {0..<V}
    by( auto simp add: prod-Var)

  then have prod Var ((λi. i * V + f i) ‘ {0..<V}) =
            prod Var ((λi. i * V + g i) ‘ {0..<V})
    by (simp add: prod.reindex[OF injf] prod.reindex[OF injg])

  then have *: ((λi. i * V + f i) ‘ {0..<V}) = ((λi. i * V + g i) ‘ {0..<V})
    by (intro prod-Var-eq) auto
  have f i = g i for i
  proof (cases i < V)
    case True
      then have fiV:f i < V
        by (simp add: assms(1))
      have (i * V + f i) ∈ ((λi. i * V + g i) ‘ {0..<V})
        using True * by force
      with div-mult-self-is-m fiV pair-enc-eq permutes-nat-less show ?thesis
        unfolding image-iff
        by (metis add-diff-cancel-right' assms(2) atLeastLessThan-iff le-less-trans)
  next
    case False
    with assms show ?thesis
```

```
      by (metis atLeastLessThan-iff permutes-def)
  qed
  thus ?thesis by auto
qed

lemma perfect-matching-det:
  assumes E ⊆ {0..<V} × {0..<V}
  assumes is-matching E match
  assumes card match = V
  showsdet (adj-mat V E) ≠ 0
proof −
  have det: det (adj-mat V E) =
     (∑p | p permutes {0..<V}.
    MPoly-Type.monom (
      sum (λi.
        monomial 1 (i ∗ V + p i)) {0..<V})
    (if ∀ i < V. (i,p i) ∈ E then
      of-int (sign p)
    else 0))
    unfolding det-adj-mat
    by auto
  from is-matching-permutes[OF assms(1−3)]
  obtain p where p: p permutes {0..<V}
     ⋀i. i < V ⟹ (i, p i) ∈ E  by auto
  then have iV: i < V ⟹
    adj-mat V E $$ (i, p i) ≠ 0 for i
    unfolding adj-mat-def
    by (subst index-mat) (auto simp add: Var-not-0)
  have ∗: of-int (sign p) ∗ (∏ i = 0..<V. adj-mat V E $$ (i, p i)) ≠ 0
    by (rule ccontr) (use iV in auto)

  have det (adj-mat V E) ≠ 0
    unfolding det
    apply (subst sum-monom-0-iff)
    subgoal
      by (auto intro!: finite-permutations)
    subgoal
      by (auto intro!: sum-monomial-eq)
    subgoal
      using p by (auto intro!: sum-monomial-eq)
    done
  thus ?thesis by auto
qed

lemma det-perfect-matching:
  assumes E ⊆ {0..<V} × {0..<V}
  assumes det (adj-mat V E) ≠ 0
  obtains match where
    is-matching E match
```

```
    card match = V
proof −
  have det: det (adj-mat V E) =
    (∑ p | p permutes {0..<V}.
       of-int (sign p) ∗
       (∏ i = 0..<V.
          adj-mat V E $$ (i, p i)))
    unfolding det-def
    by auto
  then have ... ≠ 0 using assms(2) by auto
  then obtain p where p: p permutes {0..<V}
    of-int (sign p) ∗
       (∏ i = 0..<V.
          adj-mat V E $$ (i, p i)) ≠ 0
   by (metis (no-types, lifting) mem-Collect-eq sum.not-neutral-contains-not-neutral)
  have ⋀i. i < V ⟹ adj-mat V E $$ (i, p i) ≠ 0
    using p(2)
    by simp
  then have iV: ⋀i. i < V ⟹ (i, p i) ∈ E
    unfolding adj-mat-def
    by (smt (verit, best) index-mat(1) p(1) permutes-nat-less split-conv)

  define match where match = (λi. (i,p i)) ` {0..<V}
  have 1:card match = V unfolding match-def
    apply (subst card-image)
    unfolding inj-on-def by auto
  have inj-on p {0..<V}
    using p(1) permutes-inj-on
    by blast
  hence 2:inj-on fst match inj-on snd match
    unfolding match-def
    by (auto simp add: inj-on-def)
  have 3: match ⊆ E
    using iV unfolding match-def
    by auto
  show ?thesis using that 1 2 3
    unfolding is-matching-def by auto
qed

lemma has-perfect-matching-iff:
  assumes E ⊆ {0..<V} × {0..<V}
  showshas-perfect-matching V E ⟷ det (adj-mat V E) ≠ 0
 by (metis assms det-perfect-matching has-perfect-matching-def perfect-matching-det)

lemma sum-when-1:
  assumes finite S x ∈ S
  shows   (∑ xa∈S. 1 when xa = x) = 1
  by (simp add: assms(1) assms(2) when-def)
```

lemma *total-degree-monom*:
  assumes *finite S*
  shows *total-degree* (*MPoly-Type.monom* (*sum* (*λi. monomial* (*Suc 0*) *i*) *S*) *c*) =
      (*if c* = *0 then 0 else card S*)
proof −
  have ($\sum$ *x*∈*keys* (*sum* (*monomial* (*Suc 0*)) *S*). $\sum$ *xa*∈*S*. *Suc 0 when xa* = *x*) = *card S*
    using *assms* **by** (*subst keys-sum*) (*auto simp add*: *when-def*)
  thus *?thesis*
    unfolding *MPoly-Type.monom-def* **by** (*simp add*: *total-degree.abs-eq lookup-sum single.rep-eq*)
qed

lemma *total-degree-geI*:
  assumes *m* ∈ *keys* (*mapping-of p*) ($\sum$ *v*∈*keys m. lookup m v*) ≥ *n*
  shows *total-degree p* ≥ *n*
proof −
  have *n* ≤ *Max* (*insert 0* ((*λx. sum* (*lookup x*) (*keys x*)) ' *keys* (*mapping-of p*)))
    using *assms* **by** (*subst Max-ge-iff*) *auto*
  thus *?thesis*
    by (*simp add*: *total-degree-def*)
qed

lemma *total-degree-0-iff*: *total-degree p* = *0* ⟷ *vars p* = {}
proof
  assume *total-degree p* = *0*
  show *vars p* = {}
  proof *safe*
    fix *x* **assume** *x* ∈ *vars p*
    then obtain *m* **where** *m*: *m* ∈ *keys* (*mapping-of p*) *lookup m x* > *0*
      unfolding *vars-def* **by** *blast*
    from *m* **have** *x* ∈ *keys m*
      by (*simp add*: *in-keys-iff*)
    note *m*(*2*)
    also have *lookup m x* ≤ ($\sum$ *v*∈*keys m. lookup m v*)
      by (*rule member-le-sum*) (*use* ‹*x* ∈ *keys m*› **in** *auto*)
    also have . . . ≤ *total-degree p*
      by (*intro total-degree-geI*) (*use m*(*1*) **in** *auto*)
    finally show *x* ∈ {}
      using ‹*total-degree p* = *0*› **by** *simp*
  qed
next
  assume *vars p* = {}
  hence *keys* (*mapping-of p*) ⊆ {*0*}
    by (*simp add*: *subset-eq vars-def*)
  also have (*λm. sum* (*lookup m*) (*keys m*)) ' {*0*} = {*0*}
    by *auto*
  finally have ∗: *insert 0* ((*λm. sum* (*lookup m*) (*keys m*)) ' *keys* (*mapping-of p*)) = {*0*}

by *fast*
show *total-degree p = 0*
unfolding *total-degree-def map-fun-def id-def o-def* ∗ **by** *simp*
qed

lemma *total-degree-0E*: *total-degree p = 0* $\Longrightarrow$ ($\bigwedge$*c. p = Const c* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
unfolding *total-degree-0-iff* **using** *vars-empty-iff*[*of p*] **by** *blast*

lemma *total-degree-ex*:
assumes *p* ≠ *0*
shows ∃ *m. m* ∈ *keys* (*mapping-of p*) ∧ (∑ *v*∈*keys m. lookup m v*) = *total-degree p*
proof (*cases total-degree p = 0*)
case *True*
thus *?thesis*
using *assms* **by** (*auto elim*!: *total-degree-0E simp*: *Const.rep-eq keys-Const*$_0$)
next
case *False*
have *total-degree p* ∈ *insert 0* ((λ*x. sum* (*lookup x*) (*keys x*)) ‘ *keys* (*mapping-of p*))
unfolding *total-degree-def map-fun-def o-def id-def* **by** (*rule Max-in*) *auto*
with *False* **show** *?thesis*
by *auto*
qed

lemma *coeff-times-const-left* [*simp*]: *MPoly-Type.coeff* (*Const c* ∗ *p*) *m* = *c* ∗ *MPoly-Type.coeff p m*
by (*metis Symmetric-Polynomials.coeff-smult smult-conv-mult-Const*)

lemma *total-degree-times-const-left*: *total-degree* (*Const c* ∗ *p*) ≤ *total-degree p*
proof (*cases Const c* ∗ *p = 0*)
case *False*
then obtain *m* **where** *m*:
m ∈ *keys* (*mapping-of* (*Const c* ∗ *p*)) *sum* (*lookup m*) (*keys m*) = *total-degree* (*Const c* ∗ *p*)
using *total-degree-ex* **by** *blast*
have *MPoly-Type.coeff* (*Const c* ∗ *p*) *m* ≠ *0*
using *m*(*1*) *coeff-keys* **by** *blast*
hence *MPoly-Type.coeff p m* ≠ *0*
by *auto*
hence *m* ∈ *keys* (*mapping-of p*)
using *coeff-keys* **by** *blast*
with *m*(*2*) **show** *?thesis*
by (*intro total-degree-geI*[*of m*]) *auto*
qed *auto*

lemma *total-degree-of-Const* [*simp*]: *total-degree* (*Const x*) = *0*
by (*simp add*: *total-degree-0-iff*)

```
lemma total-degree-of-int [simp]: total-degree (of-int x) = 0
  by (metis monom-of-int mpoly-monom-0-eq-Const total-degree-of-Const)

lemma total-degree-det-adj-mat: total-degree (det (adj-mat V E)) ≤ V
proof −
  have fp:finite {p. p permutes {0..<V}}
    by (intro finite-permutations) auto
  have det: det (adj-mat V E) =
    (∑ p | p permutes {0..<V}.
    MPoly-Type.monom (
      sum (λi.
        monomial (Suc 0) (i ∗ V + p i)) {0..<V})
    (if ∀ i < V. (i,p i) ∈ E then
      of-int (sign p)
    else 0)) (is - = (∑ p | p permutes {0..<V}. ?f p))
    unfolding det-adj-mat
    by auto

  have ∗: total-degree (?f p) ≤ V if p: p permutes {0..<V} for p
  proof −
    have inj: inj-on (λi. i ∗ V + p i) {0..<V}
      unfolding atLeastLessThan-iff inj-on-def
      by (metis permutes-nat-less pair-enc-eq atLeastLessThan-iff p permutes-def)
    have total-degree (?f p) =
          total-degree
                (MPoly-Type.monom (sum (monomial (Suc 0)) ((λi. i ∗ V + p
i)‘{0..<V}))
            (if ∀ i<V. (i, p i) ∈ E then sign p else 0))
      by (subst sum.reindex[OF inj]) auto
    also have . . . ≤ V
      by (subst total-degree-monom) (simp-all add: card-image[OF inj])
    finally show ?thesis .
  qed

  have total-degree (det(adj-mat V E)) ≤
    Max ((total-degree ∘ ?f) ‘{p. p permutes{0..<V}})
    unfolding det
    using total-degree-sum[OF fp, of ?f]
    by auto
  moreover have ... ≤ V
    apply (intro Max.boundedI)
    subgoal
      using fp by blast
    subgoal
      unfolding permutes-def by force
    subgoal
      using ∗ by force
    done
  ultimately show ?thesis
```

```
    by auto
qed

lemma arith:
  assumes i < V
  assumes j < (V::nat)
  shows i ∗ V + j < V^2
  by (smt (verit, ccfv-SIG) div-eq-0-iff add.right-neutral assms(1)
        assms(2) div-less-iff-less-mult div-mult-self3 le-add1 le-add-same-cancel1
        order-trans-rules(21) power2-eq-square)

lemma vars-det-adj-mat:
  showsvars (det (adj-mat V E)) ⊆ {0..<V^2}
proof −
  have det: det (adj-mat V E) =
    (∑ p | p permutes {0..<V}.
      of-int (sign p) ∗
      (∏ i = 0..<V.
          adj-mat V E $$ (i, p i)))
    unfolding det-def
    by auto
  have ∗: ⋀p. p permutes {0..<V} ⟹
    vars (of-int (sign p) ∗
      (∏ i = 0..<V.
          adj-mat V E $$ (i, p i))) ⊆ {0..<V^2}
  proof −
    fix p
    assume p permutes {0..<V}
    then have ∗: i < V ⟹
      vars (adj-mat V E $$ (i, p i)) ⊆ {0..<V^2} for i
      unfolding adj-mat-def using arith
      by (auto simp add: vars-Var)

    have vars (of-int (sign p) ∗
      (∏ i = 0..<V. adj-mat V E $$ (i, p i))) ⊆
      vars (∏ i = 0..<V. adj-mat V E $$ (i, p i))
      using vars-mult vars-signof by blast
    moreover have ... ⊆ (⋃ i∈{0..<V}. vars (adj-mat V E $$ (i, p i)))
      using vars-prod by auto
    moreover have ... ⊆ {0..<V^2} using ∗
      by fastforce
    ultimately show  vars (of-int (sign p) ∗
      (∏ i = 0..<V.
          adj-mat V E $$ (i, p i))) ⊆ {0..<V^2} by auto
  qed
  have vars (det(adj-mat V E)) ⊆
    (⋃ p ∈ {p. p permutes {0..<V}}.
      vars (of-int (sign p) ∗
      (∏ i = 0..<V.
```

```
        adj-mat V E $$ (i, p i))))
    unfolding det-def
   by (simp add: vars-sum)
  thus ?thesis using *
   by auto
qed

definition int-adj-mat::
  (nat ⇒ int) ⇒
  nat ⇒
  (nat × nat) set ⇒
  int mat
  where int-adj-mat f V E =
   mat V V (λ(i,j).
     if (i,j) ∈ E then f (i* V + j) else 0)

lemma map-mat-prod-def:
  showsmap-mat f A ≡
   Matrix.mat (dim-row A) (dim-col A)
    (λ(i,j). f (A $$ (i,j)))
  by (smt (verit, best) cong-mat map-mat-def split-conv)

lemma int-adj-mat:
  showsint-adj-mat f V E =
   map-mat (insertion f) (adj-mat V E)
  unfolding adj-mat-def int-adj-mat-def
   map-mat-prod-def
  by auto

lemma det-int-adj-mat:
  showsdet(int-adj-mat f V E) =
   insertion f (det (adj-mat V E))
  unfolding int-adj-mat
  by (subst comm-ring-hom.hom-det) (auto simp add:comm-ring-hom-insertion)

definition test-perfect-matching :: int ⇒ nat ⇒ (nat × nat) set ⇒ bool pmf
  where test-perfect-matching n V E =
    do {
      f ← Pi-pmf ({0..<V^2}) 0 (λi. pmf-of-set {0::int..<n});
      return-pmf (det (int-adj-mat f V E) ≠ 0)
    }

theorem test-perfect-matching-false-positive:
  assumes E ⊆ {0..<V} × {0..<V}
  assumes ¬has-perfect-matching V E
  shows  pmf (test-perfect-matching n V E) True = 0
proof −
  have det (adj-mat V E) = 0
   using assms(1) assms(2) has-perfect-matching-iff by blast
```

thus *?thesis*
unfolding *test-perfect-matching-def pmf-eq-0-set-pmf det-int-adj-mat*
by *auto*
qed

lemma *test-perfect-matching-true-negative*:
assumes $E \subseteq \{0..<V\} \times \{0..<V\}$
assumes $\neg$*has-perfect-matching V E*
shows *pmf* (*test-perfect-matching n V E*) *False* = *1*
by (*metis assms*(*1*) *assms*(*2*) *pmf-True-conv-False right-minus-eq test-perfect-matching-false-positive*)

theorem *test-perfect-matching-false-negative*:
assumes (*n*::*nat*) > *0*
assumes $E \subseteq \{0..<V\} \times \{0..<V\}$
assumes *has-perfect-matching V E*
shows *pmf* (*test-perfect-matching n V E*) *False* $\leq$ *V* / *n*
proof −
have *d*: *det* (*adj-mat V E*) $\neq$ *0*
using *assms has-perfect-matching-iff* **by** *blast*
have *pmf* (*test-perfect-matching n V E*) *False* =
prob-space.prob (*test-perfect-matching n V E*) {*False*}
by (*simp add*: *pmf.rep-eq*)
moreover have ... =
prob-space.prob
(*map-pmf* (λf. *det* (*int-adj-mat f V E*) $\neq$ *0*)
(*Pi-pmf* $\{0..<V^2\}$ *0* (λi. *pmf-of-set* $\{0..<int\ n\}$)))
{*False*}
unfolding *test-perfect-matching-def map-pmf-def*
by *auto*
moreover have ... =
prob-space.prob
(*Pi-pmf* $\{0..<V^2\}$ *0* (λi. *pmf-of-set* $\{0..<int\ n\}$))
{*f*. *insertion f* (*det* (*adj-mat V E*)) = *0*}
unfolding *measure-map-pmf vimage-def det-int-adj-mat*
by *auto*
moreover have ... $\leq$ *real V* / *card*$\{0..<int\ n\}$
by (*intro schwartz-zippel*[*OF - - - total-degree-det-adj-mat d vars-det-adj-mat*])
(*auto simp add*: *assms*(*1*−*2*))
moreover have ... $\leq$ *V* / *n*
using *assms*(*1*) **by** *force*
ultimately show *?thesis* **by** *auto*
qed

end

References

[1] R. A. DeMillo and R. J. Lipton. A probabilistic remark on algebraic program testing. *Inf. Process. Lett.*, 7(4):193–195, 1978.

[2] R. Lipton. The curious history of the Schwartz-Zippel lemma, 2009. Accessed on Apr 21, 2023.

[3] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.

[4] R. Zippel. Probabilistic algorithms for sparse polynomials. In E. W. Ng, editor, *EUROSAM*, volume 72 of *LNCS*, pages 216–226. Springer.