NP-hardness of CVP and SVP

Katharina Kreuzer

March 19, 2025

Abstract

This article formalizes the NP-hardness proofs of the Closest Vector Problem (CVP) and the Shortest Vector Problem (SVP) in maximum norm as well as the CVP in any p-norm for $p \geq 1$. CVP and SVP are two fundamental problems in lattice theory. Lattices are a discrete, additive subgroup of $\mathbb{R}^n$ and are used for lattice-based cryptography. The CVP asks to find the nearest lattice vector to a target. The SVP asks to find the shortest non-zero lattice vector.

This entry formalizes the basic properties of lattices, the reduction from CVP to Subset Sum in both maximum and p-norm for $1 \leq p < \infty$ and the reduction of SVP to Partition using the Bounded Homogeneous Linear Equations problem (BHLE) as an intermediate step. The formalization uncovered a number of problems with the existing proofs in the literature [7] and [6].

Contents

1 Introduction **3**

2 Reduction Function **3**

3 Basics on Lattices **4**

4 Change of Basis in a Lattice **6**

5 Partition Problem **14**

6 Subset Sum Problem **15**

7 CVP in ℓ_p for $p \geq 1$ **16**

8 Maximum Norm **19**

9 CVP in ℓ_∞ **19**

10 Representation of Integers in Different Number Systems **22**

11 Additional Lemmas **24**

12 Bounded Homogeneous Linear Equation Problem **27**

13 SVP in ℓ_∞ **30**

1 Introduction

Two problems form the very basis for cryptography on lattices, namely the closest vector problem (CVP) and the shortest vector problem (SVP). Given a finite set of basis vectors in $\mathbb{R}^n$, the set of all linear combinations with integer coefficients forms a lattice. In optimization form, SVP asks for the shortest vector in the lattice and CVP asks for the lattice vector closest to some given target vector, both with respect to some given norm. The proof of NP-hardness of these problems in ℓ_∞ goes back to an early technical report by van Emde-Boas [7]. Indeed, for other norms (especially for the Euclidean norm), NP-hardness of the SVP could only be shown using a randomized reduction [1]. For CVP, NP-hardness has been shown in any p-norm for $p \geq 1$. One exemplary proof can be found in the book by Micciancio and Goldwasser [6, Chapter 3, Thm 3.1].

The CVP and SVP are not only used in cryptography, but in various other areas as well. For example, they closely relate to integer programs (see Lenstra's paper [5]) or finding irreducible factors of polynomials as in Lenstra's paper [3]. For example, the LLL-algorithm by Lenstra, Lenstra and Lovász [4] gives a polynomial-time algorithm for lattice basis reduction which solves integer linear programs in fixed dimensions. Using this reduced basis, one can find good approximations to CVP using Babai's algorithm [2] for certain approximation factors. Still, for arbitrary dimensions, the problem remains NP-hard.

theory *Reduction*

imports
  Main
begin

2 Reduction Function

This definition was taken from the developements at https://github.com/wimmers/poly-reductions "Karp21/Reductions.thy". TODO: When this repo comes into the AFP, link to original definition.

definition *is_reduction :: "('a ⇒ 'b) ⇒ 'a set ⇒ 'b set ⇒ bool"* **where**
  "is_reduction f A B ≡ ∀a. a ∈ A ⟷ f a ∈ B "
end
theory *Lattice_int*

imports
  "Jordan_Normal_Form.Matrix"
  "Jordan_Normal_Form.VS_Connect"

"BenOr_Kozen_Reif.More_Matrix"
begin

3 Basics on Lattices

Connect the type *vec* to records of rings, commutative rings, fields and modules in order to use properties of modules such as *lin_indpt* (linear independence).

lemma *dim_carrier: "dim_vec z = dim_col A ⟹ A *$_v$ z ∈ carrier_vec (dim_row A)"*
⟨*proof*⟩

Concrete type conversion *int vec* to *real vec*

definition *real_of_int_vec :: "int vec ⇒ real vec"* **where**
"real_of_int_vec v = map_vec real_of_int v"

definition *real_to_int_vec :: "real vec ⇒ int vec"* **where**
"real_to_int_vec v = map_vec floor v"

lemma *dim_vec_real_of_int_vec[simp]: "dim_vec (real_of_int_vec v) = dim_vec v"*
⟨*proof*⟩

lemma *real_of_int_vec_nth[simp, intro]:*
"i<dim_vec v ⟹ (real_of_int_vec v) $ i = real_of_int (v$i)"
⟨*proof*⟩

lemma *real_of_int_vec_vec:*
"real_of_int_vec (vec n f) = vec n (real_of_int ∘ f)"
⟨*proof*⟩

Concrete type conversion *int mat* to *real mat*

definition *real_of_int_mat :: "int mat ⇒ real mat"* **where**
"real_of_int_mat A = map_mat real_of_int A"

definition *real_to_int_mat :: "real mat ⇒ int mat"* **where**
"real_to_int_mat A = map_mat floor A"

lemma *dim_col_real_of_int_mat[simp]: "dim_col (real_of_int_mat A) = dim_col A"*
⟨*proof*⟩

lemma *dim_row_real_of_int_mat[simp]: "dim_row (real_of_int_mat A) = dim_row A"*
⟨*proof*⟩

lemma *real_of_int_mat_nth[simp, intro]:*
"i<dim_row A ⟹ j<dim_col A ⟹ (real_of_int_mat A) $$ (i,j) = real_of_int (A $$ (i,j))"
⟨*proof*⟩

lemma *real_of_int_mat_mat:*
"real_of_int_mat (mat n m f) = mat n m (real_of_int ∘ f)"
⟨*proof*⟩

lemma *real_of_int_mat_cols:*
"cols (real_of_int_mat A) = map real_of_int_vec (cols A)"
⟨*proof*⟩

lemma *distinct_cols_real_of_int_mat:*
"distinct (cols A) = distinct (cols (real_of_int_mat A))"
⟨*proof*⟩

Algebraic lattices are discrete additive subgroups of $\mathbb{R}^n$. Lattices can be represented by a basis, multiple bases can represent the same lattice.

type_synonym *int_lattice = "int vec set"*

Linear independence

consts *is_indep :: "'a ⇒ bool"*

overloading
is_indep_real ≡ "is_indep :: real mat ⇒ bool"
is_indep_int ≡ "is_indep :: int mat ⇒ bool"
begin
definition *is_indep_real :: "real mat ⇒ bool"* **where**
*"is_indep A = (∀z::real vec. (A *$_v$ z = 0$_v$ (dim_row A) ∧ dim_vec z = dim_col A) ⟶ z = 0$_v$ (dim_vec z))"*
definition *is_indep_int :: "int mat ⇒ bool"* **where**
*"is_indep A = (∀z::real vec. ((real_of_int_mat A) *$_v$ z = 0$_v$ (dim_row A) ∧ dim_vec z = dim_col A) ⟶ z = 0$_v$ (dim_vec z))"*
end

Definition of lattices

definition *is_lattice :: "int_lattice ⇒ bool"* **where**
*"is_lattice L ≡ (∃B::(int mat). L = {B *$_v$ z | z::int vec. dim_vec z = dim_col B} ∧ is_indep B)"*

The lattice generated by the column vectors of a matrix. This matrix does not need to be linearly independent. Make certain that the output is indeed a lattice and not the entire space.

definition *gen_lattice :: "int mat ⇒ int vec set"* **where**

```
"gen_lattice A = {A *v z | z::int vec. dim_vec z = dim_col A}"
```

Theorems for lattices. The aim is to have a change of basis theorem for lattice bases. The theorems are mostly taken from the respective theorems for the type *('a,'k::finite) vec* and adapted to the type *'a vec*.

lemma *finsum_vec_carrier:*
assumes *"f ‘ A ⊆ carrier_vec nr" "finite A"*
shows *"finsum_vec TYPE('a::ring) nr f A ∈ carrier_vec nr"*
⟨*proof*⟩

lemma *dim_vec_finsum_vec:*
assumes *"f ‘ A ⊆ carrier_vec nr" "finite A"*
shows *"dim_vec (finsum_vec TYPE('a::ring) nr f A) = nr"*
⟨*proof*⟩

Matrix-Vector-Multiplication as an element in the column span.

lemma *mat_mult_as_col_span:*
assumes *"(A :: 'a :: comm_ring mat) ∈ carrier_mat nr nc"*
  and *"(z :: 'a vec) ∈ carrier_vec nc"*
shows *"A *v z = finsum_vec TYPE('a) nr (λi. z$i ·v col A i) {0..<nc}"*
  (**is** *"?left = ?right"*)
⟨*proof*⟩

A matrix is identical to the matrix generated by its indices.

lemma *mat_index:*
"B = mat (dim_row B) (dim_col B) (λ(i,j). B $$ (i,j))"
⟨*proof*⟩

Lemmas about the colums of a matrix.

lemma *col_unit_vec:*
assumes *"i<dim_col (B:: 'a :: {monoid_mult, semiring_0, zero_neq_one} mat)"*
shows *"B *v (unit_vec (dim_col B) i) = col B i"* (**is** *"?left = ?right"*)
⟨*proof*⟩

lemma *is_indep_not_null:*
assumes *"is_indep (B:: real mat)" "i<dim_col B"*
shows *"col B i ≠ 0v (dim_row B)"*
⟨*proof*⟩

lemma *col_in_gen_lattice:*
assumes *"i<dim_col B"*
shows *"col B i ∈ gen_lattice B"*
⟨*proof*⟩

4 Change of Basis in a Lattice

Definition of the column span.

definition *span :: "('a :: semiring_0) mat ⇒ 'a vec set"* **where**
*"span B = {B *$_v$ z | z:: 'a vec. dim_vec z = dim_col B}"*

Definition of the column dimension of a matrix

definition *dim :: "int mat ⇒ nat"* **where**
"dim B = (if ∃b. is_indep (real_of_int_mat b) ∧ span (real_of_int_mat b) = span (real_of_int_mat B)
then dim_col (SOME b. is_indep (real_of_int_mat b) ∧
span (real_of_int_mat b) = span (real_of_int_mat B)) else 0)"

Abbreviation of the set of columns

definition *set_cols[simp]: "set_cols A = set (cols A)"*

For the change of basis, we need to be able to delete and insert a column vector in the column span.

definition *insert_col:*
"insert_col A c = mat_of_cols (dim_row A) (c # cols A)"

definition *delete_col:*
"delete_col A c = mat_of_cols (dim_row A) (filter (λ x. ¬ (x = c)) (cols A))"

lemma *set_cols_col:*
"set_cols A = col A ` {0..<dim_col A}"
⟨*proof*⟩

lemma *set_cols_subset_col:*
assumes *"set_cols A ⊆ set_cols B"*
"i<dim_col A"
obtains *j* **where** *"col A i = col B j"*
⟨*proof*⟩

Monotonicity of the span.

lemma *span_mono:*
assumes *"set_cols (A ::'a::comm_ring mat) ⊆ set_cols B" "dim_row A = dim_row B"*
"distinct (cols A)"
shows *"span A ⊆ span B"*
⟨*proof*⟩

Monotonicity of linear independence

lemma *is_indep_mono:*
assumes *"set_cols B ⊆ set_cols A" "is_indep A" "distinct (cols B)" "dim_row A = dim_row B"*
shows *"is_indep (B::real mat)"*
⟨*proof*⟩

Linear independent vectors are distinct.

lemma *is_indep_distinct:*
assumes *"is_indep (A::real mat)"*
shows *"distinct (cols A)"*
⟨*proof*⟩

Column vectors are in the column span.

lemma *span_base:*
assumes *"(a :: 'a :: {monoid_mult,semiring_0,zero_neq_one} vec) ∈ set_cols S"*
shows *"a ∈ span S"*
⟨*proof*⟩

Representing vectors in the column span of a linear independent matrix. The representation function returns the coefficients needed to generate the vector as an element in the column span.

Respresentation function and its lemmas were adapted from "HOL/Modules.thy"

definition *representation :: "real mat ⇒ real vec ⇒ real vec ⇒ real"*
  where *"representation B v =*
    (if is_indep B ∧ v ∈ (span B) then
      SOME f. (∀ v. f v ≠ 0 ⟶ v ∈ set_cols B) ∧
        *B *$_v$ (vec (dim_col B) (λi. f (col B i))) = v*
    else (λb. 0))"

If the column vectors form a basis of the column span, the representation function is unique.

lemma *unique_representation:*
  assumes *basis: "is_indep (basis::real mat)"*
    and *in_basis: "⋀v. f v ≠ 0 ⟹ v ∈ set_cols basis" "⋀v. g v ≠ 0 ⟹ v ∈ set_cols basis"*
    and *eq: "basis *$_v$ (vec (dim_col basis) (λi. f (col basis i))) =*
          *basis *$_v$ (vec (dim_col basis) (λi. g (col basis i)))"*
  shows *"f = g"*
⟨*proof*⟩

More properties on the representation function.

lemma
  shows *representation_ne_zero: "⋀b. representation basis v b ≠ 0 ⟹ b ∈ set_cols basis"*
    and *sum_nonzero_representation_eq:*
      "is_indep basis ⟹ v ∈ span basis ⟹
      *basis *$_v$ (vec (dim_col basis) (λi. representation basis v (col basis i))) = v"*
⟨*proof*⟩

lemma *representation_eqI:*

assumes *basis: "is_indep basis"* **and** *b: "v ∈ span basis"*
and *ne_zero: "⋀b. f b ≠ 0 ⟹ b ∈ set_cols basis"*
and *eq: "basis *$_v$ (vec (dim_col basis) (λi. f (col basis i))) = v"*
shows *"representation basis v = f"*
⟨*proof*⟩

Representation function when extending the column space.

lemma *representation_extend:*
assumes *basis: "is_indep basis"* **and** *v: "v ∈ span basis'"*
and *basis': "set_cols basis' ⊆ set_cols basis"* **and** *d: "dim_row basis = dim_row basis'"*
and *dc: "distinct (cols basis')"*
shows *"representation basis v = representation basis' v"*
⟨*proof*⟩

The representation of the i-th column vector is the i-th unit vector.

lemma *representation_basis:*
assumes *basis: "is_indep basis"* **and** *b: "b ∈ set_cols basis"*
shows *"representation basis b = (λv. if v = b then 1 else 0)"*
⟨*proof*⟩

The spanning and independent set of vectors is a basis.

lemma *spanning_subset_independent:*
assumes *BA: "set_cols B ⊆ set_cols A"* **and** *iA: "is_indep (A::real mat)"*

and *AsB:"set_cols A ⊆ span B"*
and *d: "distinct (cols B)"* **and** *dr: "dim_row A = dim_row B"*
shows *"set_cols A = set_cols B"*
⟨*proof*⟩

lemma *nth_lin_combo:*
assumes *"i < dim_row A" "dim_col A = dim_vec b"*
shows *"(A *$_v$ b) $ i = (∑j=0..<dim_col A. A $$ (i,j) * b $ j)"*
⟨*proof*⟩

Insert and delete with the span.

lemma *dim_col_insert_col:*
"dim_col (insert_col S a) = dim_col S + 1"
⟨*proof*⟩

lemma *dim_col_delete_col:*
assumes *"a ∈ set_cols S" "distinct (cols S)" "dim_vec a = dim_row S"*

shows *"dim_col (delete_col S a) = dim_col S - 1"*
⟨*proof*⟩

lemma *dim_row_delete_col:*

```
  "dim_row (delete_col T b) = dim_row T"
```
⟨*proof*⟩

lemma *span_insert1:*
assumes *"dim_vec (a :: 'a ::comm_ring vec) = dim_row S"*
shows *"span (insert_col S a) = {x. ∃k y. x = y + k ·$_v$ a ∧ y ∈ span S ∧ dim_vec x = dim_row S}"*
⟨*proof*⟩

lemma *span_insert2:*
assumes *"dim_vec a = dim_row S"*
shows *"span (insert_col S (a :: 'a ::comm_ring vec)) =*
{x. ∃k. (x - k ·$_v$ a) ∈ span S ∧ dim_vec x = dim_row S}"
⟨*proof*⟩

lemma *insert_delete_span:*
assumes *"(a :: 'a ::comm_ring vec) ∈ set_cols A" "distinct (cols A)" "dim_vec a = dim_row A"*
shows *"span (insert_col (delete_col A a) a) = span (A)"*
⟨*proof*⟩

Deleting a generating element.

lemma *span_breakdown:*
assumes *bS: "(b :: 'a :: comm_ring vec) ∈ set_cols S"*
and *aS: "a ∈ span S"*
and *b: "dim_vec b = dim_row S"*
and *d: "distinct (cols S)"*
shows *"∃k. a - k ·$_v$ b ∈ span (delete_col S b)"*
⟨*proof*⟩

lemma *uminus_scalar_mult:*
"(-k) ·$_v$ a = - (k ·$_v$ (a:: 'a ::comm_ring_1 vec))"
⟨*proof*⟩

lemma *span_scale: "(x :: 'a ::field vec) ∈ span S ⟹ c ·$_v$ x ∈ span S"*
⟨*proof*⟩

lemma *span_diff:*
assumes *"(x :: 'a::ring vec) ∈ span S" "y ∈ span S"*
"x ∈ carrier_vec (dim_row S)" "y ∈ carrier_vec (dim_row S)"
shows *"x - y ∈ span S"*
⟨*proof*⟩

Adding a generating element.

lemma *in_span_insert:*

assumes *a: "(a :: 'a ::field vec) ∈ span (insert_col S b)"*
and *na: "a ∉ span S"*
and *br: "dim_vec b = dim_row S"*
and *bnotS: "b ∉ set_cols S"*
and *d: "distinct (cols S)"*
shows *"b ∈ span (insert_col S a)"*
⟨*proof*⟩

lemma *delete_not_in_set_cols:*
assumes *"b ∉ set_cols T"*
shows *"delete_col T b = T"*
⟨*proof*⟩

lemma *delete_col_span:*
assumes *"(a :: 'a ::comm_ring vec) ∈ span (delete_col T b)" "distinct (cols T)"*
shows *"a ∈ span T"*
⟨*proof*⟩

Changing a generating element.

lemma *in_span_delete_field:*
assumes *aT:"(a :: 'a :: field vec) ∈ span T"* **and** *aTb: "a ∉ span (delete_col T b)"*
and *dr: "dim_vec b = dim_row T"* **and** *d:"distinct (cols T)"*
shows *"b ∈ span (insert_col (delete_col T b) a)"*
⟨*proof*⟩

Simplifications on multiplication.

lemma *mat_of_cols_mult_vCons:*
assumes *"dim_vec (x :: 'a ::comm_ring vec) = dim_row S" "dim_vec zs = dim_col S"*
shows *"mat_of_cols (dim_row S) (x # cols S) *$_v$ vCons z0 zs = z0 ·$_v$ x + S *$_v$ zs"*
⟨*proof*⟩

lemma *mult_mat_vec_ring:*
assumes *m: "(A::'a::comm_ring mat) ∈ carrier_mat nr nc"* **and** *v: "v ∈ carrier_vec nc"*
shows *"A *$_v$ (k ·$_v$ v) = k ·$_v$ (A *$_v$ v)"* (**is** *"?l = ?r"*)
⟨*proof*⟩

Adding a span element to the generating set does not change the span.

lemma *span_redundant:*
assumes *"(x :: 'a ::comm_ring vec) ∈ span S"*
shows *"span (insert_col S x) = span S"*
⟨*proof*⟩

Transitivity of inserting elements.

lemma *span_trans:*
assumes *"(x :: 'a ::comm_ring vec) ∈ span S" "y ∈ span (insert_col S x)"*
shows *"y ∈ span S"*
⟨*proof*⟩

More on inserting, deleting and the set of columns.

lemma *delet_col_not_in_set_cols:*
assumes *"dim_vec b = dim_row T"*
shows *"b ∉ set_cols (delete_col T b)"*
⟨*proof*⟩

lemma *dim_col_distinct:*
assumes *"distinct (cols S)"*
shows *"card (set_cols S) = dim_col S"*
⟨*proof*⟩

lemma *set_cols_mono:*
assumes *"set_cols S ⊆ set_cols T" "distinct (cols S)" "distinct (cols T)"*
shows *"dim_col S ≤ dim_col T"*
⟨*proof*⟩

lemma *set_cols_insert_col:*
assumes *"dim_vec b = dim_row U"*
shows *"set_cols (insert_col U b) = set_cols U ∪ {b}"*
⟨*proof*⟩

lemma *set_cols_real_of_int_mat:*
"set_cols (real_of_int_mat S) = real_of_int_vec ‘ (set_cols S)"
⟨*proof*⟩

lemma *real_of_int_mat_span:*
"real_of_int_vec ‘ (span S) ⊆ span (real_of_int_mat S)"
⟨*proof*⟩

lemma *real_of_int_vec_ex:*
assumes *"x ∈ real_of_int_vec ‘ A"*
shows *"∃y. x = real_of_int_vec y"*
⟨*proof*⟩

lemma *real_of_int_vec_obtain:*
assumes *"x ∈ real_of_int_vec ‘ A"*
obtains *y* **where** *"x = real_of_int_vec y"*
⟨*proof*⟩

lemma *real_of_int_vec_inj:*

```
"inj real_of_int_vec"
```
⟨*proof*⟩

lemma *real_of_int_mat_mat_of_cols:*
assumes *"∀i<length cs. cs ! i ∈ carrier_vec nr"*
shows *"real_of_int_mat (mat_of_cols nr cs) = mat_of_cols nr (map real_of_int_vec cs)"*
⟨*proof*⟩

lemma *real_of_int_mat_delete_col:*
"delete_col (real_of_int_mat T) (real_of_int_vec b) = real_of_int_mat (delete_col T b)"
⟨*proof*⟩

Lemma to exchange a subset of columns from one basis to another. Pay attention that this only works over the reals, not over the integers since we need to do divisions. The *exchange_lemma* was adapted from "HOL/Vector_Spaces.thy". Note that the connection "Jordan_Normal_Form/VS_Connect" between the type *('a,'k::finite) vec* and *'a vec* does not include the theorem *exchange_lemma*. due to some problems in lifting/transfer.

lemma *exchange_lemma:*
  assumes *i: "is_indep (S::real mat)"*
    and *sp: "set_cols S ⊆ span T"*
    and *d: "distinct (cols T)"*
    and *dr: "dim_row S = dim_row T"*
  shows *"∃t'. dim_col t' = dim_col T ∧ set_cols S ⊆ set_cols t' ∧*
            set_cols t' ⊆ set_cols S ∪ set_cols T ∧
            distinct (cols t')"

⟨*proof*⟩

A linearly independent set has smaller or equal cardinality than the span it lies in.

lemma *independent_span_bound:*
  assumes *i: "is_indep (S :: int mat)"* **and** *d: "distinct (cols T)"*
    and *sp: "set_cols S ⊆ span T"*
    and *dr: "dim_row S = dim_row T"*
  shows *"dim_col S ≤ dim_col T"*
⟨*proof*⟩

When two bases B and B' generate the same lattice, both have the same length because the change of basis theorem allows us to convert one basis in the other.

lemma *gen_lattice_in_span:*
assumes *"gen_lattice B = gen_lattice B'"*
shows *"set_cols B ⊆ Lattice_int.span B'"*
⟨*proof*⟩

lemma *basis_exchange:*
  assumes *gen_eq: "gen_lattice B = gen_lattice B'"*
    and *"is_indep B"* **and** *"is_indep B'"*
  shows *"dim_col B = dim_col B'"*
⟨*proof*⟩

Basis matrix of a lattice

definition *basis_of :: "int vec set ⇒ int mat"* **where**
  "basis_of L = (SOME B. L = gen_lattice B ∧ is_indep B)"

definition *dim_lattice :: "int_lattice ⇒ nat"* **where**
  "dim_lattice L = (THE x. x = dim_col (basis_of L))"

lemma *dim_lattice_gen_lattice:*
  assumes *"is_indep B"*
  shows *"dim_lattice (gen_lattice B) = dim_col B"*
⟨*proof*⟩

A lattice generated by a linearly independent matrix is indeed a lattice.

lemma *is_lattice_gen_lattice:*
  assumes *"is_indep A"*
  shows *"is_lattice (gen_lattice A)"*
⟨*proof*⟩

end
theory *Partition*

imports
  Main
  Reduction
begin

5 Partition Problem

The Partition Problem is a widely known NP-hard problem. TODO: Reduction proof to SAT

definition *is_partition :: "int list ⇒ nat set ⇒ bool"* **where**
  "is_partition a I = (I ⊆ {0..<length a} ∧ (∑ i∈I. a ! i) = (∑ i∈({0..<length a}-I). a ! i))"

definition *partition_problem :: "(int list) set "* **where**
  "partition_problem = {a. ∃ I. I ⊆ {0..<length a} ∧
        (∑ i∈I. a ! i) = (∑ i∈({0..<length a}-I). a ! i)}"

definition *partition_problem_nonzero :: "(int list) set "* **where**
"partition_problem_nonzero = {a. ∃I. I ⊆ {0..<length a} ∧ length a > 0 ∧
(∑i∈I. a ! i) = (∑i∈({0..<length a}-I). a ! i)}"

end
theory *Subset_Sum*

imports
Main
Partition
"Jordan_Normal_Form.Matrix"
begin

6 Subset Sum Problem

The Subset Sum Problem is a common NP-hard Problem.

definition *subset_sum :: "((int vec) * int) set"* **where**
"subset_sum ≡ {(as,s). (∃xs::int vec.
(∀i<dim_vec xs. xs$i ∈ {0,1}) ∧ xs · as = s ∧ dim_vec xs = dim_vec as)}"

definition *subset_sum_nonzero :: "((int vec) * int) set"* **where**
"subset_sum_nonzero ≡ {(as,s). (∃xs::int vec.
(∀i<dim_vec xs. xs$i ∈ {0,1}) ∧ xs · as = s ∧ dim_vec xs = dim_vec as) ∧ dim_vec as ≠ 0}"

definition *subset_sum_list :: "((int list) * int) set"* **where**
"subset_sum_list ≡ {(as,s). (∃xs::int list.
*(∀i<length xs. xs!i ∈ {0,1}) ∧ (∑i<length as. as!i * xs!i) = s ∧ length xs = length as)}"*

Reduction Subset Sum to Partition.

definition *reduce_subset_sum_partition:: "(int list) ⇒ ((int list) * int)"*
where
"reduce_subset_sum_partition ≡
(λ a. (a, (if sum_list a mod 2 = 0 then sum_list a div 2 else (∑i<length a. |a!i|) + 1)))"

Well-definedness of reduction function

lemma *sum_list_map_2:*
assumes *"length a = length xs"*
shows *"(sum_list (map2 (*) a xs)) = (∑i<length a. a!i * xs!i)"*

⟨*proof*⟩

lemma *ex_01_list:*
  assumes *"I⊆{0..<n}"*
  shows *"∃xs::int list. (∀i∈I. xs ! i = 1) ∧ (∀i∈{0..<n}-I. xs ! i = 0) ∧ length xs = n"*
⟨*proof*⟩

lemma *sum_list_even:*
  assumes *"a ∈ partition_problem"*
  shows *"sum_list a mod 2 = 0"*
⟨*proof*⟩

lemma *well_defined_reduction_subset_sum:*
  assumes *"a ∈ partition_problem"*
  shows *"reduce_subset_sum_partition a ∈ subset_sum_list"*
⟨*proof*⟩

NP-hardness of reduction function

lemma *NP_hardness_reduction_subset_sum:*
  assumes *"reduce_subset_sum_partition a ∈ subset_sum_list"*
  shows *"a ∈ partition_problem"*
⟨*proof*⟩

The Subset Sum on lists is NP-hard.

lemma *"is_reduction reduce_subset_sum_partition partition_problem subset_sum_list"*
⟨*proof*⟩

Subset Sum on vectors is the same as on lists

lemma *subset_sum_vec_list:*
"(as,s) ∈ subset_sum ⟷ (list_of_vec as,s) ∈ subset_sum_list"
⟨*proof*⟩

end
theory *CVP_p*

imports
  Main
  Reduction
  Lattice_int
  Subset_Sum
begin

7 CVP in ℓ_p for $p \geq 1$

This file provides the reduction proof of Subset Sum to CVP in ℓ_p. Proof can be easily instantiated for any $p \geq 1$ using the locale variables.

definition *pth_p_norm_vec :: "nat ⇒ ('a::{abs, power, comm_monoid_add}) vec ⇒ 'a"* **where**
"pth_p_norm_vec p v = (∑ i<dim_vec v. |v$i|^p)"

locale *fixed_p =*
fixes *p::nat*
assumes *p_def: "p≥1"*
begin

definition *"p_norm_vec ≡ pth_p_norm_vec p"*

The CVP in ℓ_p

definition *is_closest_vec :: "int_lattice ⇒ int vec ⇒ int vec ⇒ bool"* **where**
"is_closest_vec L b v ≡ (is_lattice L) ∧
(∀x∈L. p_norm_vec (x - b) ≥ p_norm_vec (v - b) ∧ v∈L)"

The decision problem associated with solving CVP exactly.

definition *gap_cvp :: "(int_lattice × int vec × real) set"* **where**
"gap_cvp ≡ {(L, b, r). (is_lattice L) ∧ (∃ v∈L. of_int (p_norm_vec (v - b)) ≤ r^p)}"

Reduction function for Subset Sum to CVP in euclidean norm

definition *gen_basis_p :: "int vec ⇒ int mat"* **where**
"gen_basis_p as = mat (dim_vec as + 1) (dim_vec as) (λ (i, j). if i = 0 then as$j
else (if i = j + 1 then 2 else 0))"

definition *gen_t_p :: "int vec ⇒ int ⇒ int vec"* **where**
"gen_t_p as s = vec (dim_vec as + 1) ((λ i. 1)(0:= s))"

definition *reduce_cvp_subset_sum_p ::*
*"((int vec) * int) ⇒ (int_lattice * (int vec) * real)"* **where**
"reduce_cvp_subset_sum_p ≡ (λ (as,s).
(gen_lattice (gen_basis_p as), gen_t_p as s, root p (dim_vec as)))"

Lemmas for Proof

lemma *vec_lambda_eq[intro]: "(∀ i<n. a i = b i) ⟶ vec n a = vec n b"*
⟨*proof*⟩

lemma *eq_fun_applic:* **assumes** *"x = y"* **shows** *"f x = f y"*
⟨*proof*⟩

lemma *sum_if_zero:*

assumes *"finite A" "i∈A"*
shows *"(∑ j∈A. (if i = j then a j else 0)) = a i"*
⟨*proof*⟩

lemma *set_compr_elem:*
assumes *"finite A" "a∈A"*
shows *"{f i | i. i∈A} = {f a} ∪ {f i | i. i∈A-{a}}"*
⟨*proof*⟩

lemma *Bx_rewrite:*
assumes *x_dim: "dim_vec as = dim_vec x"*
shows *"(gen_basis_p as) *$_v$ x =*
vec (dim_vec as + 1) (λ i. if i = 0 then (x · as)
*else (2 * x$(i-1)))"*
(is *"?init_vec = ?goal_vec")*
⟨*proof*⟩

lemma *Bx_s_rewrite:*
assumes *x_dim: "dim_vec as = dim_vec x"*
shows *"(gen_basis_p as) *$_v$ x - (gen_t_p as s) =*
*vec (dim_vec as + 1) (λ i. if i = 0 then (x · as - s) else (2 * x$(i-1) - 1))"*
(is *"?init_vec = ?goal_vec")*
⟨*proof*⟩

lemma *p_norm_vec_Bx_s:*
assumes *x_dim: "dim_vec as = dim_vec x"*
shows *"p_norm_vec ((gen_basis_p as) *$_v$ x - (gen_t_p as s)) =*
*|x · as - s|^p + (∑ i=1..<dim_vec as +1. |2*x$(i-1)-1|^p)"*
⟨*proof*⟩

gen_basis_p actually generates a basis which spans the *int_lattice* (by definition) and is linearly independent.

lemma *is_indep_gen_basis_p:*
"is_indep (gen_basis_p as)"
⟨*proof*⟩

Well-definedness of the reduction function.

lemma *well_defined_reduction:*
assumes *"(as, s) ∈ subset_sum"*
shows *"reduce_cvp_subset_sum_p (as, s) ∈ gap_cvp"*
⟨*proof*⟩

NP-hardness of reduction function.

lemma *NP_hardness_reduction:*

assumes *"reduce_cvp_subset_sum_p (as, s) ∈ gap_cvp"*
shows *"(as, s) ∈ subset_sum"*
⟨*proof*⟩

CVP is NP-hard in p-norm.

lemma *"is_reduction reduce_cvp_subset_sum_p subset_sum gap_cvp"*
⟨*proof*⟩

end

end
theory *infnorm*

imports
Main
"Jordan_Normal_Form.Matrix"
"LLL_Basis_Reduction.Norms"
begin

8 Maximum Norm

The ℓ_∞ norm on vectors is exactly the maximum of the absolute value of all entries.

lemma *linf_norm_vec_Max:*
"‖v‖$_\infty$ = Max (insert 0 {|v$i| | i. i< dim_vec v})"
⟨*proof*⟩

end
theory *CVP_vec*

imports
Main
Reduction
Lattice_int
Subset_Sum
infnorm
begin

9 CVP in ℓ_∞

The closest vector problem.

definition *is_closest_vec :: "int_lattice ⇒ int vec ⇒ int vec ⇒ bool"*
where
"is_closest_vec L b v ≡ (is_lattice L) ∧
(∀x∈L. linf_norm_vec (x - b) ≥ linf_norm_vec (v - b) ∧ v∈L)"

The decision problem associated with solving CVP exactly.

definition *gap_cvp :: "(int_lattice × int vec × int) set"* **where**
"gap_cvp ≡ {(L, b, r). (is_lattice L) ∧ (∃ v∈L. linf_norm_vec (v - b) ≤ r)}"

Reduction function for CVP to subset sum

definition *gen_basis :: "int vec ⇒ int mat"* **where**
"gen_basis as = mat (dim_vec as + 2) (dim_vec as) (λ (i, j). if i ∈ {0,1} then as$j
else (if i = j + 2 then 2 else 0))"

definition *gen_t :: "int vec ⇒ int ⇒ int vec"* **where**
"gen_t as s = vec (dim_vec as + 2) ((λ i. 1)(0:= s+1, 1:= s-1))"

definition *reduce_cvp_subset_sum ::*
*"((int vec) * int) ⇒ (int_lattice * (int vec) * int)"* **where**
"reduce_cvp_subset_sum ≡ (λ (as,s).
(gen_lattice (gen_basis as), gen_t as s, (1::int)))"

Lemmas for Proof

lemma *vec_lambda_eq[intro]: "(∀ i<n. a i = b i) ⟶ vec n a = vec n b"*
⟨*proof*⟩

lemma *eq_fun_applic:* **assumes** *"x = y"* **shows** *"f x = f y"*
⟨*proof*⟩

lemma *sum_if_zero:*
assumes *"finite A" "i∈A"*
shows *"(∑ j∈A. (if i = j then a j else 0)) = a i"*
⟨*proof*⟩

lemma *set_compr_elem:*
assumes *"finite A" "a∈A"*
shows *"{f i | i. i∈A} = {f a} ∪ {f i | i. i∈A-{a}}"*
⟨*proof*⟩

lemma *Bx_rewrite:*
assumes *x_dim: "dim_vec as = dim_vec x"*
shows *"(gen_basis as) *$_v$ x =*
vec (dim_vec as + 2) (λ i. if i ∈ {0,1} then (x · as)
*else (2 * x$(i-2)))"*
(is *"?init_vec = ?goal_vec")*
⟨*proof*⟩

lemma *Bx_s_rewrite:*

assumes *x_dim: "dim_vec as = dim_vec x"*
shows *"(gen_basis as) *$_v$ x - (gen_t as s) =*
vec (dim_vec as + 2) (λ i. if i = 0 then (x · as - s - 1) else (
*if i = 1 then (x · as - s + 1) else (2 * x$(i-2) - 1)))"*
(is *"?init_vec = ?goal_vec")*
⟨*proof*⟩

lemma *linf_norm_vec_Bx_s:*
assumes *x_dim: "dim_vec as = dim_vec x"*
shows *"linf_norm_vec ((gen_basis as) *$_v$ x - (gen_t as s)) =*
Max (insert 0 ({ |x · as - s - 1|} ∪ { |x · as - s + 1|} ∪
*{ |2*x$(i-2)-1| | i. 1<i ∧ i<dim_vec as+2 }))"*
⟨*proof*⟩

gen_basis actually generates a basis which is spans the *int_lattice* (by definition) and is linearly independent.

lemma *is_indep_gen_basis:*
"is_indep (gen_basis as)"
⟨*proof*⟩

The CVP is NP-hard in ℓ_∞.

lemma *well_defined_reduction:*
assumes *"(as, s) ∈ subset_sum"*
shows *"reduce_cvp_subset_sum (as, s) ∈ gap_cvp"*
⟨*proof*⟩

NP-hardness part of reduction.

lemma *NP_hardness_reduction:*
assumes *"reduce_cvp_subset_sum (as, s) ∈ gap_cvp"*
shows *"(as, s) ∈ subset_sum"*
⟨*proof*⟩

The CVP is NP-hard in ℓ_∞.

lemma *"is_reduction reduce_cvp_subset_sum subset_sum gap_cvp"*
⟨*proof*⟩

end
theory *Digits_int*
imports
Complex_Main
begin

10 Representation of Integers in Different Number Systems

This file is an adaption of *Van_der_Waerden/Digits.thy* for representation of the Integers (instead of the natural numbers) in different number systems. The main difference is the integer input in the function *digit*.

First, we look at some useful lemmas for splitting sums.

lemma *split_sum_first_elt_less:* **assumes** *"n<m"*
shows *"(∑ i∈{n..<m}. f i) = f n + (∑ i∈{Suc n ..<m}. f i)"*
⟨*proof*⟩

lemma *split_sum_mid_less:* **assumes** *"i<(n::nat)"*
shows *"(∑ j<n. f j) = (∑ j<i. f j) + (∑ j=i..<n. f j)"*
⟨*proof*⟩

In order to use representation of numbers in a basis *base* and to calculate the conversion to and from integers, we introduce the following locale.

locale *digits =*
fixes *base :: int*
assumes *base_pos: "base > 0"*
begin

Conversion from basis base to integers: *from_digits n d*

n:	*nat*	length of representation in basis base
d:	*nat ⇒ nat*	function of digits in basis base where *d i* is the i-th digit in basis base
output:	*nat*	natural number corresponding to $d(n-1)\ldots d(0)$ as integer

fun *from_digits :: "nat ⇒ (nat ⇒ nat) ⇒ nat"* **where**
"from_digits 0 d = 0"
*| "from_digits (Suc n) d = d 0 + nat base * from_digits n (d ∘ Suc)"*

Alternative definition using sum:

lemma *from_digits_altdef: "from_digits n d = (∑ i<n. d i * nat base ^ i)"*
⟨*proof*⟩

lemma *int_from_digits:*
*"int (from_digits n d) = (∑ i<n. int (d i) * base ^ i)"*
⟨*proof*⟩

Digit in basis base of some integer number: *digit x i*

x:	*int*	integer
i:	*nat*	index
output:	*int*	i-th digit of representation in basis base of x

fun *digit :: "int ⇒ nat ⇒ int"* **where**
  "digit x 0 = |x| mod base"
| "digit x (Suc i) = digit (|x| div base) i"

Alternative definition using divisor and modulo:

lemma *digit_altdef: "digit x i = (|x| div (base ^ i)) mod base"*
⟨*proof*⟩

Every digit must be smaller that the base.

lemma *digit_less_base: "digit x i < base"*
  ⟨*proof*⟩

A representation in basis *base* of length n must be less than $base^n$.

lemma *from_digits_less:*
  assumes *"∀i<n. d i < base"*
  shows *"from_digits n d < base ^ n"*
⟨*proof*⟩

Lemmas for *mod* and *div* in number systems of basis *base*:

lemma *mod_base:* **assumes** *"⋀i. i<n ⟹ d i < base" "n>0"*
  shows *"from_digits n d mod base = d 0 "*
⟨*proof*⟩

lemma *mod_base_i:*
  assumes *"⋀i. i<n ⟹ (d i ::nat) < base" "n>0" "i<n"*
  shows *"(∑j=i..<n. d j * base ^ (j-i)) mod base = d i "*
⟨*proof*⟩

lemma *div_base_i:*
  assumes *"⋀i. i<n ⟹ (d i::nat) < base" "n>0" "i<n"*
  shows *"from_digits n d div (base ^i) = (∑j=i..<n. d j * base ^ (j-i))"*
  ⟨*proof*⟩

Conversions are inverse to each other.

lemma *digit_from_digits:*
  assumes *"⋀j. j<n ⟹ d j < base" "n>0" "i<n"*
  shows *"digit (from_digits n d) i = d i"*
  ⟨*proof*⟩

lemma *div_distrib:* **assumes** *"i<n"*
  shows *"(a*base^n + b) div base^i mod base = b div base^i mod base"*
⟨*proof*⟩

lemma(**in** *digits*) *digits_eq_0:*
  assumes *"x = 0"*
  shows *"digit x i = 0"*

⟨*proof*⟩
end

lemma *split_digits_eq_zero:*
  assumes *"a + base * b = 0" "|a|<base" "(base::int)>2"*
  shows *"a = 0 ∧ b=0"*
⟨*proof*⟩

lemma *respresentation_in_basis_eq_zero:*
  assumes *"(∑ i<n. c i * base^i) = 0" "(base::int) > 2" "⋀i. i<n ⟹ |c i| < base" "i<n"*
  shows *"c i = 0"*
⟨*proof*⟩

end
theory *Additional_Lemmas*

imports
  Main
  infnorm
  Partition
  Lattice_int
  Digits_int
begin

11 Additional Lemmas

Lemma about length and concat.

lemma *length_concat_map_5:*
  *"length (concat (map (λi. [f1 i, f2 i, f3 i, f4 i, f5 i]) xs)) = length xs * 5"*
⟨*proof*⟩

Lemma about splitting the index of sums.

lemma *sum_split_idx_prod:*
  *"(∑ i=0..<k*l::nat. f i) = (∑ i=0..<k. (∑ j=0..<l. f (i*l+j)))"*
⟨*proof*⟩

Helping lemma to split sums.

lemma *lt_M:*
  assumes *"M > (∑ i<(n::nat). |b i|::int)"*
          "∀ i<n. |x i| ≤ 1"
  shows *"|(∑ i<n. x i * b i)| < M"*
⟨*proof*⟩

lemma *split_sum:*
"(∑i<(n::nat). x i * (a i + M * b i)::int) = (∑i<n. x i * a i) + M * (∑i<n. x i * b i)"
⟨*proof*⟩

lemma *split_eq_system:*
assumes "M > (∑i<n::nat. |a i|::int)"
"∀i<n. |x i| ≤ 1"
"(∑i<n. x i * (a i + M * b i)) = 0"
shows "(∑i<n. x i * a i) = 0 ∧ (∑i<n. x i * b i) = 0"
⟨*proof*⟩

Lemmas about splitting into 4 or 5 cases.

Split into 4 modulo classes

lemma *lt_4_split:* "(i::nat) < 4 ⟶ i = 0 ∨ i = 1 ∨ i = 2 ∨ i = 3"
⟨*proof*⟩

lemma *mod_exhaust_less_4_int:* "(i::int) mod 4 = 0 ∨ i mod 4 = 1 ∨ i mod 4 = 2 ∨ i mod 4 = 3"
⟨*proof*⟩

lemma *mod_4_choices:*
assumes "i mod 4 = 0 ⟶ P i"
"i mod 4 = 1 ⟶ P i"
"i mod 4 = 2 ⟶ P i"
"i mod 4 = 3 ⟶ P i"
shows "P (i::nat)"
⟨*proof*⟩

lemma *mod_4_if_split:*
assumes "i mod 4 = 0 ⟶ P = P0 i"
"i mod 4 = 1 ⟶ P = P1 i"
"i mod 4 = 2 ⟶ P = P2 i"
"i mod 4 = 3 ⟶ P = P3 i"
shows "P = (if i mod 4 = 0 then P0 i else
(if i mod 4 = 1 then P1 i else
(if i mod 4 = 2 then P2 i else P3 (i::nat))))" (**is** "?P i")
⟨*proof*⟩

Split into 5 modulo classes

lemma *lt_5_split:* "(i::nat) < 5 ⟶ i = 0 ∨ i = 1 ∨ i = 2 ∨ i = 3 ∨ i = 4"
⟨*proof*⟩

lemma *mod_exhaust_less_5_int:*

```
  "(i::int) mod 5 = 0 ∨ i mod 5 = 1 ∨ i mod 5 = 2 ∨ i mod 5 = 3 ∨ i
mod 5 = 4"
```
⟨*proof*⟩

lemma *mod_exhaust_less_5:*
```
  "(i::nat) mod 5 = 0 ∨ i mod 5 = 1 ∨ i mod 5 = 2 ∨ i mod 5 = 3 ∨ i
mod 5 = 4"
```
⟨*proof*⟩

lemma *mod_5_choices:*
```
  assumes "i mod 5 = 0 ⟶ P i"
          "i mod 5 = 1 ⟶ P i"
          "i mod 5 = 2 ⟶ P i"
          "i mod 5 = 3 ⟶ P i"
          "i mod 5 = 4 ⟶ P i"
  shows "P (i::nat)"
```
⟨*proof*⟩

lemma *mod_5_if_split:*
```
  assumes "i mod 5 = 0 ⟶ P = P0 i"
          "i mod 5 = 1 ⟶ P = P1 i"
          "i mod 5 = 2 ⟶ P = P2 i"
          "i mod 5 = 3 ⟶ P = P3 i"
          "i mod 5 = 4 ⟶ P = P4 i"
  shows "P = (if i mod 5 = 0 then P0 i else
               (if i mod 5 = 1 then P1 i else
               (if i mod 5 = 2 then P2 i else
               (if i mod 5 = 3 then P3 i else
                                    P4 (i::nat)))))" (is "?P i")
```
⟨*proof*⟩

Representation of natural number in interval using lower bound.

lemma *split_lower_plus_diff:*
```
  assumes "s ∈ {n..<m::nat}"
  obtains j where "s = n+j" and "j<m-n"
```
⟨*proof*⟩

end
theory *BHLE*

imports
```
  Main
  Additional_Lemmas
```
begin

12 Bounded Homogeneous Linear Equation Problem

definition *bhle :: "(int vec * int) set"* **where**
"bhle ≡ {(a,k). ∃x. a · x = 0 ∧ dim_vec x = dim_vec a ∧ dim_vec a > 0 ∧
x ≠ 0$_v$ (dim_vec x) ∧ ‖x‖$_\infty$ ≤ k}"

Reduction of bounded homogeneous linear equation to partition problem

For the reduction function, one defines five-tuples for every element in a. The last tuple plays an important role. It consists only of four elements in order to add constraints to the other tuples. These values are formed in a way such that they form all constraints needed for the reductions. Note, that some indices have been changed with respect to [7] to enable better indexing in the vectors/lists.

definition *b1 :: "nat ⇒ int ⇒ int ⇒ int"* **where**
*"b1 i M a = a + M * (5^(4*i-4) + 5^(4*i-3) + 5^(4*i-1))"*

definition *b2 :: "nat ⇒ int ⇒ int"* **where**
*"b2 i M = M * (5^(4*i-3) + 5^(4*i))"*

definition *b2_last :: "nat ⇒ int ⇒ int"* **where**
*"b2_last i M = M * (5^(4*i-3) + 1)"*

definition *b3 :: "nat ⇒ int ⇒ int"* **where**
*"b3 i M = M * (5^(4*i-4) + 5^(4*i-2))"*

definition *b4 :: "nat ⇒ int ⇒ int ⇒ int"* **where**
*"b4 i M a = a + M * (5^(4*i-2) + 5^(4*i-1) + 5^(4*i))"*

definition *b4_last :: "nat ⇒ int ⇒ int ⇒ int"* **where**
*"b4_last i M a = a + M * (5^(4*i-2) + 5^(4*i-1) + 1)"*

definition *b5 :: "nat ⇒ int ⇒ int"* **where**
*"b5 i M = M * (5^(4*i-1))"*

Change order of indices in [7] such that $b3$ is in last place and can be omitted in the last entry. This ensures that the weight of the solution is 1 or -1, essential for the proof of NP-hardnes.

definition *b_list :: "int list ⇒ nat ⇒ int ⇒ int list"* **where**
"b_list as i M = [b1 (i+1) M (as!i), b2 (i+1) M, b4 (i+1) M (as!i), b5 (i+1) M, b3 (i+1) M]"

definition *b_list_last :: "int list ⇒ nat ⇒ int ⇒ int list"* **where**
"b_list_last as n M = [b1 n M (last as), b2_last n M, b4_last n M (last as), b5 n M]"

definition *gen_bhle :: "int list ⇒ int vec"* **where**
"gen_bhle as = (let M = 2(∑ i<length as. |as!i|)+1; n = length as in*
vec_of_list (concat
(map (λi. b_list as i M) [0..<n-1])
@ (if n>0 then b_list_last as n M else [])))"

The reduction function.

definition *reduce_bhle_partition:: "(int list) ⇒ ((int vec) * int)"* **where**
"reduce_bhle_partition ≡ (λ a. (gen_bhle a, 1))"

Lemmas for proof

lemma *dim_vec_gen_bhle:*
assumes *"as≠[]"*
shows *"dim_vec (gen_bhle as) = 5 * (length as) - 1"*
⟨*proof*⟩

lemma *dim_vec_gen_bhle_empty:*
"dim_vec (gen_bhle []) = 0"
⟨*proof*⟩

Lemmas about length

lemma *length_b_list:*
"length (b_list a i M) = 5" ⟨*proof*⟩

lemma *length_b_list_last:*
"length (b_list_last a n M) = 4" ⟨*proof*⟩

lemma *length_concat_map_b_list:*
*"length (concat (map (λi. b_list as i M) [0..<k])) = 5 * k"*
⟨*proof*⟩

Last values of *gen_bhle*

lemma *gen_bhle_last0:*
assumes *"length as > 0"*
shows *"(gen_bhle as) $ ((length as -1) * 5) =*
b1 (length as) (2(∑ i<length as. |as!i|)+1) (last as)"*
⟨*proof*⟩

lemma *gen_bhle_last1:*
assumes *"length as > 0"*
shows *"(gen_bhle as) $ ((length as -1) * 5 + 1) =*
b2_last (length as) (2(∑ i<length as. |as!i|)+1)"*
⟨*proof*⟩

The third entry of the last tuple is omitted, thus we skip one lemma

lemma *gen_bhle_last3:*
assumes *"length as > 0"*

shows *"(gen_bhle as) $ ((length as -1) * 5 + 2) =*
b4_last (length as) (2(∑ i<length as. |as!i|)+1) (last as)"*
⟨*proof*⟩

lemma *gen_bhle_last4:*
assumes *"length as > 0"*
shows *"(gen_bhle as) $ ((length as-1) * 5 + 3) =*
b5 (length as) (2(∑ i<length as. |as!i|)+1)"*
⟨*proof*⟩

Up to last values of *gen_bhle*

lemma *b_list_nth:*
assumes *"i<length as-1" "j<5"*
shows *"concat (map (λi. b_list as i M) [0..<length as - 1]) ! (i * 5 + j) =*
b_list as i M ! j"
⟨*proof*⟩

lemma *b_list_nth0:*
assumes *"i<length as-1"*
shows *"concat (map (λi. b_list as i M) [0..<length as - 1]) ! (i * 5) =*
b_list as i M ! 0"
⟨*proof*⟩

lemma *gen_bhle_0:*
assumes *"i<length as-1"*
shows *"(gen_bhle as) $ (i * 5) =*
b1 (i+1) (2(∑ i<length as. |as!i|)+1) (as!i)"*
⟨*proof*⟩

lemma *gen_bhle_1:*
assumes *"i<length as-1"*
shows *"(gen_bhle as) $ (i * 5 + 1) =*
b2 (i+1) (2(∑ i<length as. |as!i|)+1)"*
⟨*proof*⟩

lemma *gen_bhle_4:*
assumes *"i<length as-1"*
shows *"(gen_bhle as) $ (i * 5 + 4) =*
b3 (i+1) (2(∑ i<length as. |as!i|)+1)"*
⟨*proof*⟩

lemma *gen_bhle_2:*
assumes *"i<length as-1"*
shows *"(gen_bhle as) $ (i * 5 + 2) =*
b4 (i+1) (2(∑ i<length as. |as!i|)+1) (as!i)"*
⟨*proof*⟩

lemma *gen_bhle_3:*
  assumes *"i<length as-1"*
  shows *"(gen_bhle as) $ (i * 5 + 3) =*
    b5 (i+1) (2(∑ i<length as. |as!i|)+1)"*
⟨*proof*⟩

Well-definedness of reduction function

lemma *well_defined_reduction_subset_sum:*
  assumes *"a ∈ partition_problem_nonzero"*
  shows *"reduce_bhle_partition a ∈ bhle"*
⟨*proof*⟩

NP-hardness of reduction function

lemma *NP_hardness_reduction_subset_sum:*
  assumes *"reduce_bhle_partition a ∈ bhle"*
  shows *"a ∈ partition_problem_nonzero"*
⟨*proof*⟩

The Gap-SVP is NP-hard.

lemma *"is_reduction reduce_bhle_partition partition_problem_nonzero bhle"*
⟨*proof*⟩

end
theory *SVP_vec*

imports
  BHLE
begin

13 SVP in ℓ_∞

The reduction of SVP to BHLE in l_∞ norm

The shortest vector problem.

definition *is_shortest_vec :: "int_lattice ⇒ int vec ⇒ bool"* **where**
  "is_shortest_vec L v ≡ (is_lattice L) ∧ (∀x∈L. ‖x‖∞ ≥ ‖v‖∞ ∧ v∈L)
"

The decision problem associated with solving SVP exactly.

definition *gap_svp :: "(int_lattice × int) set"* **where**
  "gap_svp ≡ {(L, r). (is_lattice L) ∧ (dim_lattice L ≥ 2) ∧
      (∃ v∈L. ‖v‖∞ ≤ r ∧ v ≠ 0v (dim_vec v))}"

Generate a lattice to solve SVP for reduction.

Here, the factor K'' from the paper [7] was changed to be $2 \cdot \mathbf{k} \cdot (k+1) \cdot \sum_i |\mathbf{a_i}|$ in order for the proofs to finish.

definition *gen_svp_basis :: "int vec ⇒ int ⇒ int mat"* **where**
"gen_svp_basis a k = mat (dim_vec a + 1) (dim_vec a + 1)
(λ (i, j). if (i < dim_vec a) ∧ (j< dim_vec a) then (if i = j then 1 else 0)
else (if (i < dim_vec a) ∧ (j ≥ dim_vec a) then 0
*else (if (i ≥ dim_vec a) ∧ (j < dim_vec a) then (k+1) * (a $ j)*
*else 2*k*(k+1)* (∑ i ∈ {0 ..< dim_vec a}. |a $ i|) +1)))"*

Reduction SVP to bounded homogeneous linear equation problem in ℓ_∞ norm.

definition *reduce_svp_bhle:: "(int vec × int) ⇒ (int_lattice × int)"*
where
"reduce_svp_bhle ≡ (λ (a, k). (gen_lattice (gen_svp_basis a k), k))"

Lemmas for proof

lemma *gen_svp_basis_mult:*
assumes *"dim_vec z = dim_vec a + 1"*
shows *"(gen_svp_basis a k) *$_v$ z = vec (dim_vec a + 1)*
*(λi. if i < dim_vec a then z$i else (k+1) * (∑ i ∈ {0 ..< dim_vec a}. z $ i * a $ i) +*
*(2*k*(k+1)* (∑ i ∈ {0 ..< dim_vec a}. |a $ i|) +1) * (z$(dim_vec a)))"*
⟨*proof*⟩

lemma *gen_svp_basis_mult_real:*
assumes *"dim_vec z = dim_vec a + 1"*
shows *"real_of_int_mat (gen_svp_basis a k) *$_v$ z = vec (dim_vec a + 1)*

*(λi. if i < dim_vec a then z$i else (k+1) * (∑ i ∈ {0 ..< dim_vec a}. z $ i * a $ i) +*
*(2*k*(k+1)* (∑ i ∈ {0 ..< dim_vec a}. |a $ i|) +1) * (z$(dim_vec a)))"*
⟨*proof*⟩

lemma *gen_svp_basis_mult_last:*
assumes *"dim_vec z = dim_vec a + 1"*
shows *"((gen_svp_basis a k) *$_v$ z) $ (dim_vec a) =*
*(k+1) * (∑ i ∈ {0 ..< dim_vec a}. z $ i * a $ i) +*
*(2*k*(k+1)* (∑ i ∈ {0 ..< dim_vec a}. |a $ i|) +1) * (z$(dim_vec a))"*
⟨*proof*⟩

The set generated by *gen_svp_basis* is indeed linearly independent.

lemma *is_indep_gen_svp_basis:*
assumes *"k>0"*
shows *"is_indep (gen_svp_basis a k)"*
⟨*proof*⟩

lemma *insert_set_comprehension:*
"insert (f x) {f i |i. i<(x::nat)} = {f i | i. i<x+1}"
⟨*proof*⟩

lemma *bhle_k_pos:*
assumes *"(a,k) ∈ bhle"*
shows *"k>0"*
⟨*proof*⟩

lemma *svp_k_pos:*
assumes *"reduce_svp_bhle (a, k) ∈ gap_svp"*
shows *"k>0"*
⟨*proof*⟩

lemma *svp_dim_vec_a:*
assumes *"reduce_svp_bhle (a, k) ∈ gap_svp"*
shows *"dim_vec a > 0"*
⟨*proof*⟩

lemma *bhle_dim_vec_a:*
assumes *"(a, k) ∈ bhle"*
shows *"dim_vec a > 0"*
⟨*proof*⟩

lemma *first_lt_second:*
assumes *"k>0"* **and** *z_le_k:"⋀i. i< dim_vec a ⟹ |z $ i| ≤ k"*
shows *"2 * |(k + 1) * (∑ i = 0..<dim_vec a. z $ i * a $ i)|*
*< (|2 * k * (k + 1) * (∑ i = 0..<dim_vec a. |a $ i|) + 1|::int)"*
⟨*proof*⟩

Well-definedness of reduction function

lemma *well_defined_reduction_svp:*
assumes *"(a,k) ∈ bhle"*
shows *"reduce_svp_bhle (a,k) ∈ gap_svp"*
⟨*proof*⟩

NP-hardness of reduction function

lemma *NP_hardness_reduction_svp:*
assumes *"reduce_svp_bhle (a,k) ∈ gap_svp"*
shows *"(a,k) ∈ bhle"*
⟨*proof*⟩

The SVP is NP-hard in ℓ_∞.

lemma *"is_reduction reduce_svp_bhle bhle gap_svp"*
⟨*proof*⟩

end

References

[1] M. Ajtai. The shortest vector problem in L2 is NP-hard for randomized reductions (extended abstract). In *Proceedings of the thirtieth annual ACM symposium on Theory of computing - STOC '98*, pages 10–19, Dallas, Texas, United States, 1998. ACM Press.

[2] L. Babai. On Lovász lattice reduction and the nearest lattice point problem. *Combinatorica*, 6:1–13, 1986.

[3] A. K. Lenstra. Lattices and factorization of polynomials. *SIGSAM Bull.*, 15:15–16, 1981.

[4] A. K. Lenstra, H. Lenstra, and L. Lovasz. Factoring polynomials with rational coefficients. *MATH. ANN*, 261:515–534, 1982.

[5] H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, 1983.

[6] D. Micciancio and S. Goldwasser. *Complexity of Lattice Problems.* Springer US, Boston, MA, 2002.

[7] P. van Emde Boas. Another NP-Complete Partition Problem and the Complexity of Computing Short Vectors in a Lattice. tech. report 81-04. Technical report, Mathematisch Instituut, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1981.