Rabin's Closest Pair of Points Algorithm

Emin Karayel   Zixuan Fan

March 17, 2025

Abstract

This entry formalizes Rabin's randomized algorithm for the closest pair of points problem with expected linear running time. Remarkable is that the best-known deterministic algorithms have super-linear running times. Hence this algorithm is one of the first known examples of randomized algorithms that outperform deterministic algorithms.

The formalization also introduces a probabilistic time monad, which builds on the existing deterministic time monad.

Contents

1 Introduction

This entry formalizes Rabin's randomized closest points algorithm [6], with expected linear run-time.

Given a sequence of points in euclidean space, the algorithm finds the pair of points with the smallest distance between them.

Remarkable is that the best known deterministic algorithm for this problem has running time $\mathcal{O}(n \log n)$ for n points [1, Section 1]. Some of them have been formalized in Isabelle by Rau and Nipkow [7, 8].

The algorithm starts by choosing a grid-distance d, and storing the points in a square-grid whose cells have that side-length.

Then it traverses the points, computing the distance of each with the points in the same (or neighboring) cells in the square grid. (Two cells are considered neighboring, if they share an edge or a vertex.)

The fundamental dilemma of the algorithm is the correct choice of d. If it is too small, then it could happen that the two closest points of the sequence are not in neighboring cells. This means d must be chosen larger or equal to the closest-point distance of the sequence. On the other hand, if d is chosen too large, it may cause too many points ending up in the same cell, which increases the running time.

The original algorithm by Rabin, chooses d by sampling $n^{2/3}$ points and using the minimum distance of those points. This can be computed using recursion (or a sub-quadratic deterministic algorithm.)

An improvement to the algorithm, has been observed in a blog-post by Richard Lipton [5]. Instead of obtaining a sub-sample of the points in the first step to chose d, he observes that it is possible to sample n independent point pairs and computing the minimum distance of the pairs. The refined algorithm is considerably simpler, avoiding the need for recursion. Similarly, the running time proof is simpler. (This entry formalizes this later version.)

In either case, the algorithm always returns the correct result with expected linear running time.

Note that, as far as I can tell, the proof of this new version has not been published. As such this entry contains an informal proof for the results in each section.

Something that should be noted is that we assume a hypothetical data structure for the square-grid, i.e., a mapping from a pair of integers identifying the cell to the points located in the cell, that can be initialized in time $\mathcal{O}(n)$ and access time proportional to the count of points in the cell (or $\mathcal{O}(1)$ if the cell is empty.) A naive implementation of such a data structure would however have unbounded intialization time, if some points are really far apart.

The above was a discussion point that was raised by Fortune and Hopcroft [3]. Later Dietzfelbinger [2] resolved the issue by providing a concrete implementation of the data structure using a hash table, with a hash function chosen randomly from a pair-wise independent family, to guarantee the presumed costs of the hypothetical data structure in expectation. However, for the sake of simplicity and consistency with Rabin's paper, we omit this implementation detail, and pretend the hypothetical data structure exists.

Note also that, even with the hash table, it would not be possible to implement the algorithm in linear time in Isabelle directly as it requires random-access arrays.

The following introduces a few primitive algorithms for the time monad, which will be followed by the construction of the probabilistic time monad, which is necessary for the verification of the expected running time. After which the algorithm will be formalized. Its properties will be verified in the following sections.

Related Work: Closely related is a recursive meshing based approach developed by Khuller and Matias [4] in 1995. Banyassady and Mulzer have given a new analysis of the expected running time [1] of Rabin's algorithm in 2007. However, this work follows Rabin's original paper.

```
theory Randomized-Closest-Pair
  imports
    HOL−Probability.Probability-Mass-Function
    Root-Balanced-Tree.Time-Monad
    Karatsuba.Main-TM
    Closest-Pair-Points.Common
begin

hide-const (open) Giry-Monad.return
```

1.1 Preliminary Algorithms in the Time Monad

Time Monad version of *min-list*.

```
fun min-list-tm :: 'a::ord list ⇒ 'a tm where
  min-list-tm (x # y # zs) =1
    do {
      r ← min-list-tm (y#zs);
      Time-Monad.return (min x r)
    } |
    min-list-tm (x#[]) =1 Time-Monad.return x |
    min-list-tm [] =1 undefined

lemma val-min-list: xs ≠ [] ⟹ val (min-list-tm xs) = min-list xs
  by (induction xs rule:induct-list012) auto

lemma time-min-list: xs ≠ [] ⟹ time (min-list-tm xs) = length xs
  by (induction xs rule:induct-list012) (simp-all)
```

Time Monad version of *remove1*.

```
fun remove1-tm :: 'a ⇒ 'a list ⇒ 'a list tm
  where
    remove1-tm x (y#ys) =1 (
      if x = y then
        return ys
      else
        remove1-tm x ys ≫= (λr. return (y#r))
    ) |
    remove1-tm x [] =1 return []
```

lemma *val-remove1*: *val* (*remove1-tm x ys*) = *remove1 x ys*
  by (*induction ys*) *simp+*

lemma *time-remove1*: *time* (*remove1-tm x ys*) ≤ *1* + *length ys*
  by (*induction ys*) (*simp-all*)

The following is a substitute for accounting for operations, where it was not possible to do directly. One reason for this is that we abstract away the data structure of the grid (an infinite 2D-table), which properly implemented, would required the use of a hash table and 2-independent hash functions. A second reason is that we need to transfer the resource usage in the bind operation of the probabilistic time monad (See below in the definition *bind-tpmf*).

fun *custom-tick* :: *nat* ⇒ *unit tm*
  where
    custom-tick (*Suc n*) =*1 custom-tick n* |
    custom-tick 0 = *return* ()

lemma *time-custom-tick*: *time* (*custom-tick n*) = *n* **by** (*induction n*) *auto*

1.2 Probabilistic Time Monad

The following defines the probabilistic time monad using the type *'a tm pmf*, i.e., the algorithm returns a probability space of pairs of values and time-consumptions.

Note that the alternative type *'a pmf tm*, i.e., a constant time consumption with a value-distribution does not work since the running time may depend on random choices.

type-synonym *'a tpmf* = *'a tm pmf*

definition *bind-tpmf* :: *'a tpmf* ⇒ (*'a* ⇒ *'b tpmf*) ⇒ *'b tpmf*
  where *bind-tpmf m f* =
    do {
      x ← *m*;
      r ← *f* (*val x*);
      return-pmf (*custom-tick* (*time x*) ≫= (λ-. *r*))
    }

definition *return-tpmf* :: *'a* ⇒ *'a tpmf*
  where *return-tpmf x* = *return-pmf* (*return x*)

The following allows the lifting of a deterministic algorithm in the time monad into the probabilistic time monad.

definition *lift-tm* :: *'a tm* ⇒ *'a tpmf*
  where *lift-tm x* = *return-pmf x*

The following allows the lifting of a randomized algorithm into the probabilisitc time monad. Note this should only be done, for primitive cases, as it requires accounting of the time usage.

definition *lift-pmf* :: *nat* ⇒ *'a pmf* ⇒ *'a tpmf*
  where *lift-pmf k m* = *map-pmf* (λ*x*. *custom-tick k* ⤜ (λ-. *return x*)) *m*

adhoc-overloading *Monad-Syntax.bind* ⇌ *bind-tpmf*

lemma *val-bind-tpmf*:
  map-pmf val (*bind-tpmf m f*) = *map-pmf val m* ⤜ (λ*x*. *map-pmf val* (*f x*))
  (**is** *?L* = *?R*)
proof −
  have *map-pmf val* (*bind-tpmf m f*) = *m* ⤜ (λ*x*. *f* (*val x*) ⤜ (λ*x*. *return-pmf* (*val x*)))
    unfolding *bind-tpmf-def map-bind-pmf* **by** *simp*
  also have ... = *?R* **unfolding** *bind-map-pmf* **by** (*simp add*: *map-pmf-def*)
  finally show *?thesis* **by** *simp*
qed

lemma *val-return-tpmf*:
  map-pmf val (*return-tpmf x*) = *return-pmf x*
  unfolding *return-tpmf-def* **by** *simp*

lemma *val-lift-tpmf*: *map-pmf val* (*lift-pmf k x*) = *x*
  unfolding *lift-pmf-def val-bind-tpmf map-pmf-comp* **by** *simp*

lemma *val-lift-tm*:
  map-pmf val (*lift-tm x*) = *return-pmf* (*val x*)
  unfolding *lift-tm-def* **by** *simp*

lemmas *val-tpmf-simps* = *val-bind-tpmf val-lift-tpmf val-return-tpmf val-lift-tm*

lemma *time-return-tpmf*: *map-pmf time* (*return-tpmf x*) = *return-pmf 0*
  unfolding *return-tpmf-def* **by** *simp*

lemma *time-lift-pmf*: *map-pmf time* (*lift-pmf x p*) = *return-pmf x*
  unfolding *lift-pmf-def map-pmf-comp* **by** (*simp add*: *time-custom-tick*)

lemma *time-bind-tpmf*: *map-pmf time* (*bind-tpmf m f*) =
  do {
    x ← *m*;
    y ← *f* (*val x*);
    return-pmf (*time x* + *time y*)
  }
  unfolding *bind-tpmf-def map-bind-pmf* **by** (*simp add*:*time-custom-tick*)

lemma *bind-return-tm*: *bind-tm* (*Time-Monad.return x*) *f* = *f x*
  by (*simp add*:*tm-simps tm.case-eq-if*)

lemma *bind-return-tpmf*: *bind-tpmf* (*return-tpmf x*) *f* = (*f x*)
unfolding *bind-tpmf-def return-tpmf-def*
by (*simp add*:*bind-return-pmf bind-return-tm bind-return-pmf'*)

Version of *replicate-pmf* for the probabilistic time monad.

fun *replicate-tpmf* :: *nat* ⇒ *'a tpmf* ⇒ *'a list tpmf*
where
replicate-tpmf 0 p = *return-tpmf* [] |
replicate-tpmf (*Suc n*) *p* =
do {
x ← *p*;
y ← *replicate-tpmf n p*;
return-tpmf (*x#y*)
}

lemma *time-replicate-tpmf*:
map-pmf time (*replicate-tpmf n p*) = *map-pmf sum-list* (*replicate-pmf n* (*map-pmf time p*))
proof (*induction n*)
case *0* **thus** *?case* **by** (*simp add*:*time-return-tpmf*)
next
case (*Suc n*)
have *map-pmf time* (*replicate-tpmf* (*Suc n*) *p*) =
p ⤜ (λ*x*. *replicate-tpmf n p* ⤜ (λ*y*. *return-pmf* (*time x* + *time y*)))
by (*simp add*: *time-bind-tpmf return-tpmf-def*)
(*simp add*: *bind-tpmf-def bind-assoc-pmf bind-return-pmf time-custom-tick*)
also have ... = *map-pmf time p* ⤜
(λ*x*. *map-pmf time* (*replicate-tpmf n p*) ⤜ (λ*y*. *return-pmf* (*x* + *y*)))
unfolding *map-pmf-def* **by** (*simp add*:*bind-assoc-pmf bind-return-pmf*)
also have ... = *map-pmf time p* ⤜ (λ*x*. *replicate-pmf n* (*map-pmf time p*) ⤜
(λ*y*. *return-pmf* (*x* + *sum-list y*)))
by (*subst Suc*) (*metis* (*no-types*, *lifting*) *bind-map-pmf bind-pmf-cong*)
also have ... = *map-pmf sum-list* (*replicate-pmf* (*Suc n*) (*map-pmf time p*))
by (*simp add*:*map-bind-pmf*)
finally show *?case* **by** *simp*
qed

lemma *val-replicate-tpmf*:
map-pmf val (*replicate-tpmf n x*) = *replicate-pmf n* (*map-pmf val x*)
by (*induction n*) (*simp-all add*:*val-tpmf-simps*)

lemma *set-val-replicate-tpmf*:
assumes *xs* ∈ *set-pmf* (*replicate-tpmf n p*)
shows *length* (*val xs*) = *n set* (*val xs*) ⊆ *val* ‘ *set-pmf p*
proof −
have *val xs* ∈ *set-pmf* (*map-pmf val* (*replicate-tpmf n p*)) **using** *assms* **by** *simp*
thus *length* (*val xs*) = *n* *set* (*val xs*) ⊆ *val* ‘ *set-pmf p*
unfolding *val-replicate-tpmf set-replicate-pmf* **by** *auto*

qed

lemma *replicate-return-pmf*[*simp*]: *replicate-pmf n* (*return-pmf x*) = *return-pmf* (*replicate n x*)
  by (*induction n*) (*simp-all add*:*bind-return-pmf*)

1.3 Randomized Closest Points Algorithm

Using the above we can express the randomized closests points algorithm in the probabilistic time monad.

```
type-synonym point = real^2

record grid =
  g-dist :: real
  g-lookup :: int ∗ int ⇒ point list tm

definition to-grid :: real ⇒ point ⇒ int ∗ int
  where to-grid d x = (⌊x $ 1/d⌋,⌊x $ 2/d⌋)
```

This represents the grid data-structure mentioned before. We assume the build time is linear to the number of points stored and the access time is at least $\mathcal{O}(1)$ and proportional to the number of points in the cell. (In practice this would be implemented using hash functions.)

```
definition build-grid :: point list ⇒ real ⇒ grid tm where
  build-grid xs d =
  do {
    - ← custom-tick (length xs);
    return (|
      g-dist = d,
      g-lookup = (λq. map-tm return (filter (λx. to-grid d x = q) xs))
    |)
  }

definition sample-distance :: point list ⇒ real tpmf where
  sample-distance ps = do {
    i ← lift-pmf 1 (pmf-of-set {i. fst i < snd i ∧ snd i < length ps});
    return-tpmf (dist (ps ! (fst i)) (ps ! (snd i)))
  }

lemma val-sample-distance:
  map-pmf val (sample-distance ps) = map-pmf (λi. dist (ps ! (fst i)) (ps ! (snd
i)))
  (pmf-of-set {i. fst i < snd i ∧ snd i < length ps})
 unfolding sample-distance-def by (simp add:val-tpmf-simps) (simp add:map-pmf-def)

definition first-phase :: point list ⇒ real tpmf where
  first-phase ps = do {
    ds ← replicate-tpmf (length ps) (sample-distance ps);
```

```
  lift-tm (min-list-tm ds)
}
```

```
definition lookup-neighborhood :: grid ⇒ point ⇒ point list tm
  where lookup-neighborhood grid p =
    do {
      d ← tick (g-dist grid);
      q ← tick (to-grid d p);
      cs ← map-tm (λx. tick (x + q)) [(0,0),(0,1),(1,−1),(1,0),(1,1)];
      map-tm (g-lookup grid) cs ≫= concat-tm ≫= remove1-tm p
    }
```

This function collects all points in the cell of the given point and those from the neighboring cells. Here it is relevant to note that only half of the neighboring cells are taken. This is because of symmetry, i.e., if point p is north-east of point q, then q is south-west of point q. Since all points are being traversed it is enough to restrict the neighbor set.

```
definition calc-dists-neighborhood :: grid ⇒ point ⇒ real list tm
  where calc-dists-neighborhood grid p =
    do {
      ns ← lookup-neighborhood grid p;
      map-tm (tick ∘ dist p) ns
    }
```

```
definition second-phase :: real ⇒ point list ⇒ real tm where
  second-phase d ps = do {
    grid ← build-grid ps d;
    ns ← map-tm (calc-dists-neighborhood grid) ps;
    concat-tm ns ≫= min-list-tm
  }
```

```
definition closest-pair :: point list ⇒ real tpmf where
  closest-pair ps = do {
    d ← first-phase ps;
    if d = 0 then
      lift-tm (tick 0)
    else
      lift-tm (second-phase d ps)
  }
```

end

2 Correctness

This section verifies that the algorithm always returns the correct result.

Because the algorithm checks every pair of points in the same or in neighboring cells. It is enough to establish that the grid distance is at least the distance of the closest pair.

The latter is true by construction, because the grid distance is chosen as a minimum of actually occurring point distances.

theory *Randomized-Closest-Pair-Correct*
  imports *Randomized-Closest-Pair*
begin

definition *min-dist* :: ('*a*::*metric-space*) *list* ⇒ *real*
  where *min-dist xs* = *Min* {*dist x y*|*x y*. {# *x*, *y*#} ⊆# *mset xs*}

For a list with length at least two, the result is the minimum distance between the points of any two elements of the list. This means that *min-dist xs* = *0*, if and only if the same point occurs twice in the list.

Note that this means, we won't assume the distinctness of the input list, and show the correctness of the algorithm in the above sense.

lemma *image-conv-2*: {*f x y*|*x y*. *p x y*} = (*case-prod f*) ' {(*x*,*y*). *p x y*} **by** *auto*

lemma *min-dist-set-fin*: *finite* {*dist x y*|*x y*. {#*x*, *y*#} ⊆# *mset xs*}
proof −
  have *a*:*finite* (*set xs* × *set xs*) **by** *simp*
  have *x* ∈# *mset xs* ∧ *y* ∈# *mset xs* **if** {#*x*, *y*#} ⊆# *mset xs* **for** *x y*
    using *that* **by** (*meson insert-union-subset-iff mset-subset-eq-insertD*)
  thus *?thesis* **unfolding** *image-conv-2* **by** (*intro finite-imageI finite-subset*[*OF* - *a*]) *auto*
qed

lemma *min-dist-ne*: *length xs* ≥ *2* ⟷ {*dist x y*|*x y*. {# *x*,*y*#} ⊆# *mset xs*} ≠ {} (**is** *?L* ⟷ *?R*)
proof
  assume *?L*
  then obtain *xh1 xh2 xt* **where** *xs*:*xs*=*xh1*#*xh2*#*xt* **by** (*metis Suc-le-length-iff numerals*(*2*))
  hence {#*xh1*,*xh2*#} ⊆# *mset xs* **unfolding** *xs* **by** *simp*
  thus *?R* **by** *auto*
next
  assume *?R*
  then obtain *x y* **where** *xy*: {#*x*,*y*#} ⊆# *mset xs* **by** *auto*
  have *2* ≤ *size* {#*x*, *y*#} **by** *simp*
  also have ... ≤ *size* (*mset xs*) **by** (*intro size-mset-mono xy*)
  finally have *2* ≤ *size* (*mset xs*) **by** *simp*
  thus *?L* **by** *simp*
qed
lemmas *min-dist-neI* = *iffD1*[*OF min-dist-ne*]

lemma *min-dist-nonneg*:
  assumes *length xs* ≥ *2*
  shows *min-dist xs* ≥ *0*

unfolding *min-dist-def* **by** (*intro Min.boundedI min-dist-set-fin assms iffD1*[*OF min-dist-ne*]) *auto*

lemma *min-dist-pos-iff*:
assumes *length xs* $\geq$ *2*
shows *distinct xs* $\longleftrightarrow$ *0* $<$ *min-dist xs*
proof −
have $\neg$(*distinct xs*) $\longleftrightarrow$ ($\exists$ *x. count* (*mset xs*) *x* $\neq$ *of-bool* (*x* $\in$ *set xs*))
unfolding *of-bool-def distinct-count-atmost-1* **by** *fastforce*
also have ... $\longleftrightarrow$ ($\exists$ *x. count* (*mset xs*) *x* $\notin$ {*0*,*1*})
using *count-mset-0-iff* **by** (*intro ex-cong1*) *simp*
also have ... $\longleftrightarrow$ ($\exists$ *x. count* (*mset xs*) *x* $\geq$ *count* {#*x*, *x*#} *x*)
by (*intro ex-cong1*) (*simp add*:*numeral-eq-Suc Suc-le-eq dual-order.strict-iff-order*)
also have ... $\longleftrightarrow$ ($\exists$ *x.* {#*x*, *x*#} $\subseteq$# *mset xs*) **by** (*intro ex-cong1*) (*simp add*: *subseteq-mset-def*)
also have ... $\longleftrightarrow$ *0* $\in$ {*dist x y* |*x y.* {#*x*, *y*#} $\subseteq$# *mset xs*} **by** *auto*
also have ... $\longleftrightarrow$ *min-dist xs* = *0* (**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *?L*
hence *min-dist xs* $\leq$ *0* **unfolding** *min-dist-def* **by** (*intro Min-le min-dist-set-fin*)
thus *min-dist xs* = *0* **using** *min-dist-nonneg*[*OF assms*] **by** *auto*
next
assume *?R*
thus *0* $\in$ {*dist x y* |*x y.* {#*x*, *y*#} $\subseteq$# *mset xs*}
unfolding *min-dist-def* **using** *Min-in*[*OF min-dist-set-fin min-dist-neI*[*OF assms*]] **by** *simp*
qed
finally have $\neg$(*distinct xs*) $\longleftrightarrow$ *min-dist xs* = *0* **by** *simp*
thus *?thesis* **using** *min-dist-nonneg*[*OF assms*] **by** *auto*
qed

lemma *multiset-filter-mono-2*:
assumes $\bigwedge$*x. x* $\in$ *set-mset xs* $\Longrightarrow$ *P x* $\Longrightarrow$ *Q x*
shows *filter-mset P xs* $\subseteq$# *filter-mset Q xs* (**is** *?L* $\subseteq$# *?R*)
proof −
have *?L* = *filter-mset* (λ*x. Q x* $\wedge$ *P x*) *xs* **using** *assms* **by** (*intro filter-mset-cong*) *auto*
also have ... = *filter-mset P* (*filter-mset Q xs*) **by** (*simp add*:*filter-filter-mset*)
also have ... $\subseteq$# *?R* **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *filter-mset-disj*:
filter-mset (λ*x. p x* $\vee$ *q x*) *xs* = *filter-mset* (λ*x. p x* $\wedge$ $\neg$ *q x*) *xs* + *filter-mset q xs*
by (*induction xs*) *auto*

lemma *size-filter-mset-decompose*:
assumes *finite T*
shows *size* (*filter-mset* (λ*x. f x* $\in$ *T*) *xs*) = ($\sum$ *t* $\in$ *T. size* (*filter-mset* (λ*x. f x*

$= t)\ xs))$
  using *assms*
proof (*induction T*)
  case *empty* **thus** *?case* **by** *simp*
next
  case (*insert x F*) **thus** *?case* **by** (*simp add*:*filter-mset-disj*) *metis*
qed

lemma *size-filter-mset-decompose'*:
  size (*filter-mset* ($\lambda x.\ f\ x \in T$) *xs*) = *sum'* ($\lambda t.$ *size* (*filter-mset* ($\lambda x.\ f\ x = t$) *xs*)) *T*
  (**is** *?L* = *?R*)
proof −
  let *?T* = *f* ‘ *set-mset xs* ∩ *T*
  have *?L* = *size* (*filter-mset* ($\lambda x.\ f\ x \in$ *?T*) *xs*)
    by (*intro arg-cong*[**where** *f*=*size*] *filter-mset-cong*) *auto*
  also have ... = ($\sum t \in$ *?T*. *size* (*filter-mset* ($\lambda x.\ f\ x = t$) *xs*))
    by (*intro size-filter-mset-decompose*) *auto*
  also have ... = *sum'* ($\lambda t.$ *size* (*filter-mset* ($\lambda x.\ f\ x = t$) *xs*)) *?T*
    by (*intro sum.eq-sum*[*symmetric*]) *auto*
  also have ... = *?R* **by** (*intro sum.mono-neutral-left'*) *auto*
  finally show *?thesis* **by** *simp*
qed

lemma *filter-product*:
  filter ($\lambda x.$ *P* (*fst x*)∧*Q* (*snd x*)) (*List.product xs ys*) = *List.product* (*filter P xs*) (*filter Q ys*)
proof (*induction xs*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons xh xt*) **thus** *?case* **by** (*simp add*:*filter-map comp-def*)
qed

lemma *floor-diff-bound*: $|\lfloor x\rfloor - \lfloor y\rfloor| \le \lceil |x - (y::real)| \rceil$ **by** *linarith*

lemma *power2-strict-mono*:
  fixes *x y* :: *'a* :: *linordered-idom*
  assumes $|x| < |y|$
  shows $x\hat{\ }2 < y\hat{\ }2$
  using *assms* **unfolding** *power2-eq-square*
  by (*metis abs-mult-less abs-mult-self-eq*)

definition *grid ps d* = ⦇ *g-dist* = *d*, *g-lookup* = ($\lambda q.$ *map-tm return* (*filter* ($\lambda x.$ *to-grid d x* = *q*) *ps*)) ⦈

lemma *build-grid-val*: *val* (*build-grid ps d*) = *grid ps d*
  unfolding *build-grid-def grid-def* **by** *simp*

```
lemma lookup-neighborhood:
  mset (val (lookup-neighborhood (grid ps d) p)) =
   filter-mset (λx. to-grid d x − to-grid d p ∈ {(0,0),(0,1),(1,−1),(1,0),(1,1)})
(mset ps) − {#p#}
proof −
  define ls where  ls = [(0::int,0::int),(0,1),(1,−1),(1,0),(1,1)]
  define g where g = grid ps d
 define cs where cs = map ((+) (to-grid (g-dist g) p)) ([(0,0),(0,1),(1,−1),(1,0),(1,1)])

  have distinct-ls: distinct ls unfolding ls-def by (simp add: upto.simps)

  have mset (concat (map (λx. val (g-lookup g (x + to-grid (g-dist g) p))) ls)) =
    mset (concat (map (λx. filter (λq. to-grid d q − to-grid d p = x) ps) ls))
    by (simp add:grid-def filter-eq-val-filter-tm cs-def comp-def algebra-simps ls-def
g-def)
  also have ... = {# q ∈# mset ps. to-grid d q − to-grid d p ∈ set ls #}
    using distinct-ls by (induction ls) (simp-all add:filter-mset-disj, metis)
 also have ... = {#x ∈# mset ps. to-grid d x − to-grid d p ∈ {(0,0),(0,1),(1,−1),(1,0),(1,1)}#}
    unfolding ls-def by simp
  finally have a:
    mset (concat (map (λx. val (g-lookup g (x + to-grid (g-dist g) p))) ls)) =
   {#x ∈# mset ps. to-grid d x − to-grid d p ∈ {(0,0),(0,1),(1,−1),(1,0),(1,1)}#}
by simp

  thus ?thesis
    unfolding g-def[symmetric] lookup-neighborhood-def ls-def[symmetric]
    by (simp add:val-remove1 comp-def)
qed

lemma fin-nat-pairs: finite {(i, j). i < j ∧ j < (n::nat)}
  by (rule finite-subset[where B={..<n }×{..<n}]) auto

lemma mset-list-subset:
  assumes distinct ys set ys ⊆ {..<length xs}
  shows  mset (map ((!) xs) ys) ⊆# mset xs (is ?L ⊆# ?R)
proof −
  have mset ys ⊆# mset [0..<length xs] using assms
   by (metis finite-lessThan mset-set-set mset-set-upto-eq-mset-upto subset-imp-msubset-mset-set)
  hence image-mset ((!) xs) (mset ys) ⊆# image-mset ((!) xs) (mset ([0..<length
xs]))
    by (intro image-mset-subseteq-mono)
 moreover have image-mset ((!) xs) (mset ([0..<length xs])) = mset xs by (metis
map-nth mset-map)
  ultimately show ?thesis by simp
qed

lemma sample-distance:
  assumes length ps ≥ 2
  shows AE d in map-pmf val (sample-distance ps). min-dist ps ≤ d
```

```
proof -
  let ?S = {i. fst i < snd i ∧ snd i < length ps}
  let ?p = pmf-of-set ?S

  have (0,1) ∈ ?S using assms by auto
  hence a:finite ?S ?S ≠ {}
    using fin-nat-pairs[where n=length ps] by (auto simp:case-prod-beta')

  have min-dist ps ≤ dist (ps ! (fst x)) (ps ! (snd x)) if x ∈ ?S for x
  proof -
    have mset (map ((!) ps) [fst x, snd x]) ⊆# mset ps
      using that by (intro mset-list-subset) auto
    hence {#ps ! fst x, ps ! snd x#} ⊆# mset ps by simp
    hence (λ(x, y). dist x y) (ps ! (fst x), ps ! (snd x)) ∈ {dist x y |x y. {#x, y#}
⊆# mset ps}
      unfolding image-conv-2 by (intro imageI) simp
    thus ?thesis unfolding min-dist-def by (intro Min-le min-dist-set-fin) simp
  qed
  thus ?thesis
   using a unfolding sample-distance-def map-pmf-def[symmetric] val-tpmf-simps
    by (intro AE-pmfI) (auto)
qed

lemma first-phase:
  assumes length ps ≥ 2
  shows AE d in map-pmf val (first-phase ps). min-dist ps ≤ d
proof -
  have min-dist ps ≤ val (min-list-tm ds)
   if ds-range:set ds⊆set-pmf(map-pmf val (sample-distance ps)) and length ds=length
ps for ds
  proof -
    have ds-ne: ds ≠ [] using assms that(2) by auto

    have min-dist ps ≤ a if a ∈ set ds for a
    proof -
     have a ∈ set-pmf (map-pmf val (sample-distance ps)) using ds-range that by
auto
     thus ?thesis using sample-distance[OF assms] by (auto simp add: AE-measure-pmf-iff)
    qed
    hence min-dist ps ≤ Min (set ds) using ds-ne by (intro Min.boundedI) auto
    also have ... = min-list ds unfolding min-list-Min[OF ds-ne] by simp
    also have ... = val (min-list-tm ds) by (intro val-min-list[symmetric] ds-ne)
    finally show ?thesis by simp
  qed

  thus ?thesis
    unfolding first-phase-def val-tpmf-simps val-replicate-tpmf
    by (intro AE-pmfI) (auto simp:set-replicate-pmf)
qed
```

definition *grid-lex-ord* :: *int* ∗ *int* ⇒ *int* ∗ *int* ⇒ *bool*
  where *grid-lex-ord x y* = (*fst x* < *fst y* ∨ (*fst x* = *fst y* ∧ *snd x* ≤ *snd y*))

lemma *grid-lex-order-antisym*: *grid-lex-ord x y* ∨ *grid-lex-ord y x*
  unfolding *grid-lex-ord-def* **by** *auto*

lemma *grid-dist*:
  fixes *p q* :: *point*
  assumes *d* > *0*
  shows |⌊*p* $ *k*/*d*⌋ − ⌊*q* $ *k*/*d*⌋| ≤ ⌈*dist p q*/*d*⌉
proof −
  have |*p*$*k* − *q*$*k*| = *sqrt* ((*p*$*k* − *q*$*k*)^*2*) **by** *simp*
  also have ... = *sqrt* (∑*j*∈*UNIV*. *of-bool*(*j*=*k*)∗(*p*$*j* − *q*$*j*)^*2*) **by** *simp*
  also have ... ≤ *dist p q* **unfolding** *dist-vec-def L2-set-def*
    by (*intro real-sqrt-le-mono sum-mono*) (*auto simp*:*dist-real-def*)
  finally have |*p*$*k* − *q*$*k*| ≤ *dist p q* **by** *simp*
  hence *0*:|*p*$*k* /*d* − *q*$*k* /*d*|≤ *dist p q* /*d* **using** *assms* **by** (*simp add*:*field-simps*)
  have |⌊*p*$*k*/*d*⌋ − ⌊*q*$*k*/*d*⌋| ≤ ⌈|*p*$*k* /*d* − *q*$*k* /*d*|⌉ **by** (*intro floor-diff-bound*)
  also have ... ≤ ⌈*dist p q*/*d*⌉ **by** (*intro ceiling-mono 0*)
  finally show *?thesis* **by** *simp*
qed

lemma *grid-dist-2*:
  fixes *p q* :: *point*
  assumes *d* > *0*
  assumes ⌈*dist p q*/*d*⌉ ≤ *s*
  shows *to-grid d p* − *to-grid d q* ∈ {−*s*..*s*}×{−*s*..*s*}
proof −
  have *f* (*to-grid d p*) − *f* (*to-grid d q*) ∈ {−*s*..*s*} **if** *f* = *fst* ∨ *f* = *snd* **for** *f*
  proof −
    have |*f* (*to-grid d p*) − *f* (*to-grid d q*)| ≤ ⌈*dist p q*/*d*⌉
      using *that grid-dist*[*OF assms*(*1*)] **unfolding** *to-grid-def* **by** *auto*
    also have ... ≤ *s* **by** (*intro assms*(*2*))
    finally have |*f* (*to-grid d p*) − *f* (*to-grid d q*)| ≤ *s* **by** *simp*
    thus *?thesis* **by** *auto*
  qed
  thus *?thesis* **by** (*simp add*:*mem-Times-iff*)
qed

lemma *grid-dist-3*:
  fixes *p q* :: *point*
  assumes *d* > *0*
  assumes ⌈*dist q p*/*d*⌉ ≤ *1* *grid-lex-ord* (*to-grid d p*) (*to-grid d q*)
  shows *to-grid d q* − *to-grid d p* ∈ {(*0*,*0*),(*0*,*1*),(*1*,−*1*),(*1*,*0*),(*1*,*1*)}
proof −
  have *a*:{−*1*..*1*} = {−*1*,*0*,*1*::*int*} **by** *auto*
  let *?r* = *to-grid d q* − *to-grid d p*
  have *?r* ∈ {−*1*..*1*}×{−*1*..*1*} **by** (*intro grid-dist-2 assms*(*1*−*2*))

moreover have *?r* ∉ {(−*1*,*0*),(−*1*,−*1*),(−*1*,*1*),(*0*,−*1*)} **using** *assms*(*3*)
unfolding *grid-lex-ord-def insert-iff de-Morgan-disj*
by (*intro conjI notI*) (*simp-all add*:*algebra-simps*)
ultimately show *?thesis* **unfolding** *a* **by** *simp*
qed

lemma *second-phase-aux*:
assumes *d* > *0 min-dist ps* ≤ *d length ps* ≥ *2*
obtains *u v* **where**
min-dist ps = *dist u v*
{#*u*, *v*#} ⊆# *mset ps*
grid-lex-ord (*to-grid d u*) (*to-grid d v*)
u ∈ *set ps v* ∈ *set* (*val* (*lookup-neighborhood* (*grid ps d*) *u*))
proof −
have ∃ *u v*. *min-dist ps* = *dist u v* ∧ {#*u*, *v*#} ⊆# *mset ps*
unfolding *min-dist-def* **using** *Min-in*[*OF min-dist-set-fin min-dist-neI*[*OF assms*(*3*)]] **by** *auto*

then obtain *u v* **where** *uv*:
min-dist ps = *dist u v* {#*u*, *v*#} ⊆# *mset ps*
grid-lex-ord (*to-grid d u*) (*to-grid d v*)
using *add-mset-commute dist-commute grid-lex-order-antisym* **by** (*metis* (*no-types*, *lifting*))

have *u-range*: *u* ∈ *set ps* **using** *uv*(*2*) *set-mset-mono* **by** *fastforce*

have *to-grid d v* − *to-grid d u* ∈ {(*0*,*0*),(*0*,*1*),(*1*,−*1*),(*1*,*0*),(*1*,*1*)}
using *assms*(*1*,*2*) *uv*(*1*,*3*) **by** (*intro grid-dist-3*) (*simp-all add*:*dist-commute*)

hence *v* ∈# *mset* (*val* (*lookup-neighborhood* (*grid ps d*) *u*))
using *uv*(*2*) **unfolding** *lookup-neighborhood* **by** (*simp add*: *in-diff-count insert-subset-eq-iff*)

thus *?thesis* **using** *that u-range uv* **by** *simp*
qed

lemma *second-phase*:
assumes *d* > *0 min-dist ps* ≤ *d length ps* ≥ *2*
shows *val* (*second-phase d ps*) = *min-dist ps* (**is** *?L* = *?R*)
proof −
let *?g* = *grid ps d*

have ∃ *u v*. *min-dist ps* = *dist u v* ∧ {#*u*, *v*#} ⊆# *mset ps*
unfolding *min-dist-def* **using** *Min-in*[*OF min-dist-set-fin min-dist-neI*[*OF assms*(*3*)]] **by** *auto*

then obtain *u v* **where** *uv*:
min-dist ps = *dist u v* {#*u*, *v*#} ⊆# *mset ps*
grid-lex-ord (*to-grid d u*) (*to-grid d v*)

and *u-range*: $u \in$ *set ps*
and *v-range*: $v \in$ *set* (*val* (*lookup-neighborhood* (*grid ps d*) *u*))
using *second-phase-aux*[*OF assms*] **by** *auto*

hence *a*: *val* (*lookup-neighborhood* (*grid ps d*) *u*) $\neq$ [] **by** *auto*

have $\exists$ *x*$\in$*set ps*. *min-dist ps* $\in$ *dist x* ‘ *set* (*val* (*lookup-neighborhood* (*grid ps d*) *x*))
using *v-range uv*(*1*) **by** (*intro bexI*[**where** *x*=*u*] *u-range*) *simp*

hence *b*: *Min* ($\bigcup$*x*$\in$*set ps*. *dist x* ‘ *set* (*val* (*lookup-neighborhood* (*grid ps d*) *x*))) $\leq$ *min-dist ps*
by (*intro Min.coboundedI finite-UN-I*) *simp-all*

have {# *x*, *y*#} $\subseteq$# *mset ps*
if $x \in$ *set ps* $y \in$ *set* (*val* (*lookup-neighborhood* (*grid ps d*) *x*)) **for** *x y*
proof −
have *y* $\in$# *mset* (*val* (*lookup-neighborhood* (*grid ps d*) *x*)) **using** *that* **by** *simp*
moreover have *mset* (*val* (*lookup-neighborhood* (*grid ps d*) *x*)) $\subseteq$# *mset ps* − {#*x*#}
using *that*(*1*) **unfolding** *lookup-neighborhood subset-eq-diff-conv* **by** *simp*
ultimately have *y* $\in$# *mset ps* − {#*x*#} **by** (*metis mset-subset-eqD*)
moreover have *x* $\in$# *mset ps* **using** *that*(*1*) **by** *simp*
ultimately show {#*x*, *y*#} $\subseteq$# *mset ps* **by** (*simp add*: *insert-subset-eq-iff*)
qed
hence *c*: *min-dist ps* $\leq$ *Min* ($\bigcup$*x*$\in$*set ps*. *dist x* ‘ *set* (*val* (*lookup-neighborhood* (*grid ps d*) *x*)))
unfolding *min-dist-def* **using** *a u-range* **by** (*intro Min-antimono min-dist-set-fin*) *auto*

have *?L* = *val* (*min-list-tm* (*concat* (*map* (λ*x*. *map* (*dist x*) (*val* (*lookup-neighborhood* *?g x*))) *ps*)))
unfolding *second-phase-def* **by** (*simp add*:*calc-dists-neighborhood-def build-grid-val*)
also have ... = *min-list* (*concat* (*map* (λ*x*. *map* (*dist x*) (*val* (*lookup-neighborhood* *?g x*))) *ps*))
using *assms*(*3*) *a u-range* **by** (*intro val-min-list*) *auto*
also have ... = *Min* ($\bigcup$*x*$\in$*set ps*. *dist x* ‘ *set* (*val* (*lookup-neighborhood* *?g x*)))
using *a u-range* **by** (*subst min-list-Min*) *auto*
also have ... = *min-dist ps* **using** *b c* **by** *simp*
finally show *?thesis* **by** *simp*
qed

Main result of this section:

theorem *closest-pair-correct*:
assumes *length ps* $\geq$ *2*
shows *AE r in map-pmf val* (*closest-pair ps*). *r* = *min-dist ps*
proof −
define *fp* **where** *fp* = *map-pmf val* (*first-phase ps*)

have *r* = *min-dist ps* **if**
d ∈ *fp*
r = (*if d* = *0 then 0 else val* (*second-phase d ps*)) **for** *r d*
proof −
have *d-ge*: *d* ≥ *min-dist ps*
using *that*(*1*) *first-phase*[*OF assms*] **unfolding** *AE-measure-pmf-iff fp-def*[*symmetric*] **by** *simp*
show *?thesis*
proof (*cases d* > *0*)
case *True*
thus *?thesis* **using** *second-phase*[*OF True d-ge assms*] *that*(*2*)
by (*simp add*: *AE-measure-pmf-iff*)
next
case *False*
hence *d* = *0 min-dist ps* = *0* **using** *d-ge min-dist-nonneg*[*OF assms*] **by** *auto*
then show *?thesis* **using** *that*(*2*) **by** *auto*
qed
qed
thus *?thesis* **unfolding** *closest-pair-def val-tpmf-simps fp-def*[*symmetric*] *if-distrib*
by (*intro AE-pmfI*) (*auto simp*:*if-distrib*)
qed

end

3 Growth of Close Points

This section verifies a result similar to (but more general than) Lemma 2 by Rabin [6]. Let $N(d)$ denote the number of pairs from the point sequence $p_1, \ldots, p_n$, with distance less than d:

$$N(d) := |\{(i,j)|d(p_i,p_j) < d \wedge 1 \leq i,j \leq n\}|$$

Obviously, $N(d)$ is monotone. It is possible to show that the growth of $N(d)$ is bounded.
In particular:

$$N(ad) \leq (2a\sqrt{2}+3)^2 N(d)$$

for all $a > 0$, $d > 0$. As far as we can tell the proof below is new.
Proof: Consider a 2D-grid with size $\alpha := \frac{d}{\sqrt{2}}$ and let us denote by $G(x,y)$ the number of points that fall in the cell $(x,y) \in \mathbb{Z} \times \mathbb{Z}$, i.e.:

$$G(x,y) := \left|\left\{i \middle| \left\lfloor \frac{p_{i,1}}{\alpha} \right\rfloor = x \wedge \left\lfloor \frac{p_{i,2}}{\alpha} \right\rfloor = x \right\}\right|,$$

where $p_{i,1}$ (resp. $p_{i,2}$) denote the first (resp. second) component of point p. Let also $s := \lceil a\sqrt{2} \rceil$.

Then we can observe that

$$
\begin{aligned}
N(ad) &\leq \sum_{(x,y)\in\mathbb{Z}\times\mathbb{Z}} \sum_{i=-s}^{s} \sum_{j=-s}^{s} G(x,y)G(x+i,y+j) \\
&= \sum_{i=-s}^{s} \sum_{j=-s}^{s} \sum_{(x,y)\in\mathbb{Z}\times\mathbb{Z}} G(x,y)G(x+i,y+j) \\
&\leq \sum_{i=-s}^{s} \sum_{j=-s}^{s} \left(\left(\sum_{(x,y)\in\mathbb{Z}\times\mathbb{Z}} G(x,y)^2 \right) \left(\sum_{(x,y)\in\mathbb{Z}\times\mathbb{Z}} G(x+i,y+j)^2 \right) \right)^{1/2} \\
&\leq \sum_{i=-s}^{s} \sum_{j=-s}^{s} \left(\left(\sum_{(x,y)\in\mathbb{Z}\times\mathbb{Z}} G(x,y)^2 \right) \left(\sum_{(x,y)\in\mathbb{Z}\times\mathbb{Z}} G(x,y)^2 \right) \right)^{1/2} \\
&\leq (2s+1)^2 \sum_{(x,y)\in\mathbb{Z}\times\mathbb{Z}} G(x,y)^2 \\
&\leq (2a\sqrt{(}2)+3)^2 \sum_{(x,y)\in\mathbb{Z}\times\mathbb{Z}} G(x,y)^2 \\
&\leq (2a\sqrt{(}2)+3)^2 N(d)
\end{aligned}
$$

The first inequality follows from the fact that if two points are ad close, their x-coordinates and y-coordinates will differ by at most ad. I.e. their grid coordinates will differ at most by s. This means the pair will be accounted for in the right hand side of the inequality.

The third inequality is an application of the Cauchy–Schwarz inequality.

The last inequality follows from the fact that the largest possible distance of two points in the same grid cell is d. □

theory *Randomized-Closest-Pair-Growth*
  imports
    HOL−Library.Sublist
    Randomized-Closest-Pair-Correct
begin

lemma *inj-translate*:
  fixes *a b* :: *int*
  shows *inj* (λ*x*. (*fst x* + *a*, *snd x* + *b*))
proof −
  have *0*:(λ*x*. (*fst x* + *a*, *snd x* + *b*)) = (λ*x*. *x* + (*a*,*b*)) **by** *auto*
  show *?thesis* **unfolding** *0* **by** *simp*
qed

lemma *of-nat-sum′*:
  (*of-nat* (*sum′ f S*) :: (*′a* :: {*semiring-char-0*})) = *sum′* (λ*x*. *of-nat* (*f x*)) *S*
  unfolding *sum.G-def* **by** *simp*

lemma *sum′-nonneg*:
  fixes f :: $'a \Rightarrow 'b$ :: {*ordered-comm-monoid-add*}
  assumes $\bigwedge x.\ x \in S \Longrightarrow f\ x \geq 0$
  shows $sum'\ f\ S \geq 0$
proof −
  have $0 \leq sum\ f\ \{x \in S.\ f\ x \neq 0\}$ **using** *assms* **by** (*intro sum-nonneg*) *auto*
  thus *?thesis* **unfolding** *sum.G-def* **by** *simp*
qed

lemma *sum′-mono*:
  fixes f :: $'a \Rightarrow 'b$ :: {*ordered-comm-monoid-add*}
  assumes $\bigwedge x.\ x \in S \Longrightarrow f\ x \leq g\ x$
  assumes *finite* $\{x \in S.\ f\ x \neq 0\}$
  assumes *finite* $\{x \in S.\ g\ x \neq 0\}$
  shows $sum'\ f\ S \leq sum'\ g\ S$ (**is** $?L \leq ?R$)
proof −
  let $?S = \{i \in S.\ f\ i \neq 0\} \cup \{i \in S.\ g\ i \neq 0\}$

  have $?L = sum'\ f\ ?S$ **by** (*intro sum.mono-neutral-right′*) *auto*
  also have ... $= (\sum i \in ?S.\ f\ i)$ **using** *assms* **by** (*intro sum.eq-sum*) *auto*
  also have ... $\leq (\sum i \in ?S.\ g\ i)$ **using** *assms* **by** (*intro sum-mono*) *auto*
  also have ... $= sum'\ g\ ?S$ **using** *assms* **by** (*intro sum.eq-sum*[*symmetric*]) *auto*
  also have ... $= ?R$ **by** (*intro sum.mono-neutral-left′*) *auto*
  finally show *?thesis* **by** *simp*
qed

lemma *cauchy-schwarz′*:
  assumes *finite* $\{i \in S.\ f\ i \neq 0\}$
  assumes *finite* $\{i \in S.\ g\ i \neq 0\}$
  shows $sum'\ (\lambda i.\ f\ i * g\ i)\ S \leq sqrt\ (sum'\ (\lambda i.\ f\ i\hat{}2)\ S) * sqrt\ (sum'\ (\lambda i.\ g\ i\hat{}2)\ S)$
    (**is** $?L \leq ?R$)
proof −
  let $?S = \{i \in S.\ f\ i \neq 0\} \cup \{i \in S.\ g\ i \neq 0\}$

  have $?L = sum'\ (\lambda i.\ f\ i * g\ i)\ ?S$ **by** (*intro sum.mono-neutral-right′*) *auto*
  also have ... $= (\sum i \in ?S.\ f\ i * g\ i)$ **using** *assms* **by** (*intro sum.eq-sum*) *auto*
  also have ... $\leq (\sum i \in ?S.\ |f\ i| * |g\ i|)$ **by** (*intro sum-mono*) (*metis abs-ge-self abs-mult*)
  also have ... $\leq L2\text{-}set\ f\ ?S * L2\text{-}set\ g\ ?S$ **by** (*rule L2-set-mult-ineq*)
  also have ... $= sqrt\ (sum'\ (\lambda i.\ f\ i\hat{}2)\ ?S) * sqrt\ (sum'\ (\lambda i.\ g\ i\hat{}2)\ ?S)$
    unfolding *L2-set-def* **using** *assms sum.eq-sum* **by** *simp*
  also have ... $= ?R$
    by (*intro arg-cong2*[**where** $f=(\lambda x\ y.\ sqrt\ x * sqrt\ y)$] *sum.mono-neutral-left′*) *auto*
  finally show *?thesis* **by** *simp*
qed

```
context comm-monoid-set
begin

lemma reindex-bij-betw′:
  assumes bij-betw h S T
  shows G (λx. g (h x)) S = G g T
proof −
  have h ‘ {x ∈ S. g (h x) ≠ 1} = {x ∈ T. g x ≠ 1}
    using bij-betw-imp-surj-on[OF assms] by auto
  hence 0: bij-betw h {x ∈ S. g (h x) ≠ 1} {x ∈ T. g x ≠ 1}
    by (intro bij-betw-subset[OF assms]) auto
  hence finite {x ∈ S. g (h x) ≠ 1} = finite {x ∈ T. g x ≠ 1}
    using bij-betw-finite by auto
  thus ?thesis unfolding G-def using reindex-bij-betw[OF 0] by simp
qed

end


definition close-point-size xs d = length (filter (λ(p,q). dist p q < d) (List.product
xs xs))

lemma grid-dist-upper:
  fixes p q :: point
  assumes d > 0
  shows dist p q < sqrt (∑ i∈UNIV.(d∗(|⌊p$i/d⌋−⌊q$i/d⌋|+1))^2)
    (is ?L < ?R)
proof −
  have a:|x − y| < |d ∗ real-of-int (|⌊x/d⌋ − ⌊y/d⌋| + 1)| for x y :: real
  proof −
    have |x − y| = d ∗ |x/d − y/d|
      using assms by (simp add: abs-mult-pos′ right-diff-distrib)
    also have ... < d ∗ real-of-int (|⌊x/d⌋ − ⌊y/d⌋| + 1)
      by (intro mult-strict-left-mono assms) linarith
    also have ... = |d ∗ real-of-int (|⌊x/d⌋ − ⌊y/d⌋| + 1)|
      using assms by simp
    finally show ?thesis by simp
  qed
  have ?L = sqrt (∑ i∈UNIV. (p $ i − q $ i)²)
    unfolding dist-vec-def dist-real-def L2-set-def by simp
  also have ... < ?R
    using assms by (intro real-sqrt-less-mono sum-strict-mono power2-strict-mono
a) auto
  finally show ?thesis by simp
qed

lemma grid-dist-upperI:
  fixes p q :: point
  fixes d :: real
```

assumes $d > 0$
assumes $\bigwedge k.\ |\lfloor p\$k/d\rfloor - \lfloor q\$k/d\rfloor| \le s$
shows $dist\ p\ q < d * (s+1) * sqrt\ 2$
proof −
have *s-ge-0*: $s \ge 0$ **using** *assms*(2)[**where** $k=0$] **by** *simp*
have $dist\ p\ q < sqrt\ (\sum i \in UNIV.\ (d*(|\lfloor p\$i/d\rfloor - \lfloor q\$i/d\rfloor|+1))\hat{}2)$
by (*intro grid-dist-upper assms*)
also have ... $\le sqrt\ (\sum i \in (UNIV::2\ set).\ (d*(s+1))\hat{}2)$
using *assms*
by (*intro real-sqrt-le-mono sum-mono power-mono mult-left-mono iffD2*[*OF of-int-le-iff*]) *auto*
also have ... $= sqrt\ (2 * (d*(s+1))\hat{}2)$ **by** *simp*
also have ... $= sqrt\ 2 * sqrt\ ((d*(s+1))\hat{}2)$ **by** (*simp add*:*real-sqrt-mult*)
also have ... $= sqrt\ 2 * (d * (s+1))$ **using** *assms s-ge-0* **by** *simp*
also have ... $= d * (s+1) * sqrt\ 2$ **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *close-point-approx-upper*:
fixes *xs* :: *point list*
fixes *G* :: *int* × *int* ⇒ *real*
assumes $d > 0$ $e > 0$
defines $s \equiv \lceil d\ /\ e\rceil$
defines $G \equiv (\lambda x.\ real\ (length\ (filter\ (\lambda p.\ to\text{-}grid\ e\ p = x)\ xs)))$
shows $close\text{-}point\text{-}size\ xs\ d \le (\sum i \in \{-s..s\}\times\{-s..s\}.\ sum'\ (\lambda x.\ G\ x * G\ (x+i))\ UNIV)$
(**is** $?L \le ?R$)
proof −
let $?f = to\text{-}grid\ e$
let $?pairs = mset\ (List.product\ xs\ xs)$

define *T* **where** $T = \{-s..s\} \times \{-s..s\}$

have $s \ge 1$ **unfolding** *s-def* **using** *assms* **by** *simp*
hence *s-ge-0*: $s \ge 0$ **by** *simp*

have *0*: *finite T* **unfolding** *T-def* **by** *simp*

have *a*: $size\ \{\#p \in\#\ ?pairs.\ ?f\ (fst\ p) - ?f\ (snd\ p) = i\ \#\} = sum'\ (\lambda x.\ G\ x * G\ (x+i))\ UNIV$
(**is** $?L1 = ?R1$) **for** *i*
proof −
have $?L1 = size\ \{\#p \in\#\ ?pairs.\ (?f\ (fst\ p), ?f\ (snd\ p)) \in \{(x,y).\ x - y = i\}\ \#\}$
by *simp*
also have ... $= sum'\ (\lambda q.\ size\ \{\#\ p \in\#\ ?pairs.\ (?f\ (fst\ p),\ ?f\ (snd\ p)) = q\ \#\})\ \{(x,y).\ x-y=i\}$
unfolding *size-filter-mset-decompose'* **by** *simp*
also have ... $= sum'\ (\lambda q.\ size\ \{\#\ p \in\#\ ?pairs.\ (?f\ (fst\ p),\ ?f\ (snd\ p)) =$

(*q*+*i*,*q*) #}) *UNIV*
by (*intro arg-cong*[**where** *f*=*real*] *sum.reindex-bij-betw'*[*symmetric*] *bij-betwI*[**where** *g*=*snd*])
auto
also have ... =
sum' (λ*q*. *length* (*filter* (λ*p*. *?f* (*fst p*) = *q*+*i* ∧ *?f* (*snd p*) = *q*) (*List.product xs xs*))) *UNIV*
by (*simp flip*: *size-mset mset-filter conj-commute*)
also have ... = *sum'* (λ*x*. *G* (*x*+*i*) ∗ *G x*) *UNIV*
by (*subst filter-product*)
(*simp add*:*G-def build-grid-def of-nat-sum' case-prod-beta' prod-eq-iff*)
finally show *?thesis* **by** (*simp add*:*algebra-simps*)
qed

have *b*:*f* (*?f p*) − *f* (*?f q*) ∈ {−*s*..*s*} **if** *f* = *fst* ∨ *f* = *snd dist p q* < *d* **for** *p q f*
proof −
have |*f* (*?f p*) − *f* (*?f q*)| ≤ ⌈*dist p q*/*e*⌉
using *grid-dist*[*OF assms*(*2*), **where** *p*=*p* **and** *q*=*q*] *that*(*1*) **unfolding** *to-grid-def* **by** *auto*
also have ... ≤ *s*
unfolding *s-def* **using** *that*(*2*) *assms*(*1*,*2*)
by (*simp add*: *ceiling-mono divide-le-cancel*)
finally have |*f* (*?f p*) − *f* (*?f q*)| ≤ *s* **by** *simp*
thus *?thesis* **using** *s-ge-0* **by** *auto*
qed

have *c*:*?f p* − *?f q* ∈ *T* **if** *dist p q* < *d* **for** *p q*
unfolding *T-def* **using** *b*[*OF - that*] **unfolding** *mem-Times-iff* **by** *simp*

have *?L* = *size* (*filter-mset* (λ(*p*,*q*). *dist p q* < *d*) *?pairs*)
unfolding *close-point-size-def* **by** (*metis mset-filter size-mset*)
also have ... ≤ *size* (*filter-mset* (λ*p*. *?f* (*fst p*) − *?f* (*snd p*) ∈ *T*) *?pairs*)
using *c* **by** (*intro size-mset-mono of-nat-mono multiset-filter-mono-2*) *auto*
also have ... = ($\sum$ *i* ∈ *T*. *size* (*filter-mset* (λ*p*. *?f* (*fst p*) − *?f* (*snd p*) = *i*) *?pairs*))
by (*intro size-filter-mset-decompose arg-cong*[**where** *f*=*of-nat*] *0*)
also have ... = ($\sum$ *i* ∈ *T*. *sum'* (λ*x*. *G x* ∗ *G* (*x*+*i*)) *UNIV*)
unfolding *of-nat-sum* **by** (*intro sum.cong a refl*)
also have ... = *?R* **unfolding** *T-def* **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *close-point-approx-lower*:
fixes *xs* :: *point list*
fixes *G* :: *int* × *int* ⇒ *real*
fixes *d* :: *real*
assumes *d* > *0*
defines *G* ≡ (λ*x*. *real* (*length* (*filter* (λ*p*. *to-grid d p* = *x*) *xs*)))
shows *sum'* (λ*x*. *G x* ^ *2*) *UNIV* ≤ *close-point-size xs* (*d* ∗ *sqrt 2*)

(**is** *?L* ≤ *?R*)
proof −
let *?f* = *to-grid d*
let *?pairs* = *mset* (*List.product xs xs*)

have *?L* = *sum′* (λ*x*. *length* (*filter* (λ*p*. *?f p* = *x*) *xs*)^*2*) *UNIV*
unfolding *build-grid-def G-def* **by** (*simp add*:*of-nat-sum′ prod-eq-iff case-prod-beta′*)
also have ... = *sum′*(λ*x*. *length*(*List.product* (*filter*(λ*p*. *?f p*=*x*)*xs*) (*filter*(λ*p*. *?f p*=*x*)*xs*))) *UNIV*
unfolding *length-product* **by** (*simp add*:*power2-eq-square*)
also have ... = *sum′* (λ*x*. *length* (*filter*(λ*p*. *?f*(*fst p*)=*x*∧*?f*(*snd p*)=*x*)(*List.product xs xs*))) *UNIV*
by (*subst filter-product*) *simp*
also have ... = *sum′* (λ*x*. *size* {# *p* ∈# *?pairs*. *?f* (*fst p*) = *x* ∧ *?f* (*snd p*) = *x* #}) *UNIV*
by (*intro arg-cong2*[**where** *f*=*sum′*] *arg-cong*[**where** *f*=*real*] *refl ext*)
(*metis* (*no-types*, *lifting*) *mset-filter size-mset*)
also have ... = *sum′* (λ*x*. *size* {# *p* ∈#{# *p*∈#*?pairs*. *?f*(*fst p*)=*?f*(*snd p*) #}. *?f*(*fst p*)=*x* #}) *UNIV*
unfolding *filter-filter-mset*
by (*intro sum.cong′ arg-cong*[**where** *f*=*real*] *arg-cong*[**where** *f*=*size*] *filter-mset-cong*) *auto*
also have ... = *size* {# *p* ∈# {# *p* ∈# *?pairs*. *?f* (*fst p*) = *?f* (*snd p*) #}. *?f* (*fst p*) ∈ *UNIV* #}
by (*intro arg-cong*[**where** *f*=*real*] *size-filter-mset-decompose′*[*symmetric*])
also have ... ≤ *size* {# *p* ∈# *?pairs*. *?f* (*fst p*) = *?f* (*snd p*) #} **by** *simp*
also have ... = *size* {# *p* ∈# *?pairs*. ∀ *k*. ⌊*fst p* $ *k*/*d*⌋ = ⌊*snd p* $ *k*/*d*⌋ #}
unfolding *to-grid-def prod.inject*
by (*intro arg-cong*[**where** *f*=*size*] *arg-cong*[**where** *f*=*of-nat*] *filter-mset-cong refl*)
(*metis* (*full-types*) *exhaust-2 one-neq-zero*)
also have ... ≤ *size* {# *p* ∈# *?pairs*. *dist* (*fst p*) (*snd p*) < *d* ∗ *of-int* (*0*+*1*) ∗ *sqrt 2* #}
by (*intro of-nat-mono size-mset-mono multiset-filter-mono-2 grid-dist-upperI*[*OF assms*(*1*)]) *simp*
also have ... = *?R* **unfolding** *close-point-size-def*
by (*simp add*:*case-prod-beta′*) (*metis* (*no-types*, *lifting*) *mset-filter size-mset*)
finally show *?thesis* **by** *simp*
qed

lemma *build-grid-finite*:
assumes *inj f*
shows *finite* {*x*. *filter* (λ*p*. *to-grid d p* = *f x*) *xs* ≠ []}
proof −
have *0*:*finite* (*to-grid d* ‘ *set xs*) **by** (*intro finite-imageI*) *auto*
have *finite* {*x*. *filter* (λ*p*. *to-grid d p* = *x*) *xs* ≠ []}
unfolding *filter-empty-conv* **by** (*intro finite-subset*[*OF* - *0*]) *blast*
hence *finite* (*f* −‘ {*x*. *filter* (λ*p*. *to-grid d p* = *x*) *xs* ≠ []}) **by** (*intro finite-vimageI assms*)

```
  thus ?thesis by (simp add:vimage-def)
qed
```

Main result of this section:

```
lemma growth-lemma:
  fixes xs :: point list
  assumes a > 0 d > 0
  shows close-point-size xs (a ∗ d) ≤ (2 ∗ sqrt 2 ∗ a + 3)^2 ∗ close-point-size xs
d
    (is ?L ≤ ?R)
proof −
  let ?s = ⌈a ∗ sqrt 2⌉
  let ?G = (λx. real (length (filter (λp. to-grid (d/sqrt 2) p = x)  xs)))
  let ?I = {−?s..?s}×{−?s..?s}

  have ?s ≥ 1 using assms by auto
  hence s-ge-0: ?s ≥ 0 by simp

  have a: ?s = ⌈a ∗ d / (d / sqrt 2)⌉ using assms by simp

  have ?L ≤ (∑ i∈{−?s..?s}×{−?s..?s}. sum′ (λx. ?G x ∗ ?G (x+i)) UNIV)
    using assms unfolding a by (intro close-point-approx-upper) auto
  also have ... ≤ (∑ i∈?I. sqrt (sum′ (λx. ?G x^2) UNIV) ∗ sqrt (sum′ (λx. ?G
(x+i)^2) UNIV))
   by (intro sum-mono cauchy-schwarz′) (auto intro: inj-translate build-grid-finite)
  also have ... = (∑ i∈?I. sqrt (sum′ (λx. ?G x^2) UNIV) ∗ sqrt (sum′ (λx. ?G
x^2) UNIV))
    by (intro arg-cong2[where f=(λx y. sqrt x ∗ sqrt y)] sum.cong refl
        sum.reindex-bij-betw′ bij-plus-right)
  also have ... = (∑ i∈?I. |sum′ (λx. ?G x^2) UNIV|) by simp
  also have ... = (2∗ ?s + 1)^2 ∗ |sum′ (λx. ?G x^2) UNIV|
    using s-ge-0 by (auto simp: power2-eq-square)
  also have ... = (2∗ ?s + 1)^2 ∗ sum′ (λx. ?G x^2) UNIV
    by (intro arg-cong2[where f=(∗)] refl abs-of-nonneg sum′-nonneg) auto
  also have ... ≤ (2∗?s+1)^2 ∗ real (close-point-size xs ((d/sqrt 2)∗ sqrt 2))
    using assms by (intro mult-left-mono close-point-approx-lower) auto
  also have ... =  (2 ∗ of-int ?s+1)^2 ∗ real (close-point-size xs d) by simp
  also have ... ≤ (2 ∗ (a ∗ sqrt 2 + 1) + 1)^2 ∗ real (close-point-size xs d)
   using s-ge-0 by (intro mult-right-mono power-mono add-mono mult-left-mono)
auto
  also have ... = ?R by (auto simp:algebra-simps)
  finally show ?thesis by simp
qed

end
```

4 Speed

In this section, we verify that the running time of the algorithm is linear with respect to the length of the point sequence $p_1, \ldots, p_n$.

Proof: It is easy to see that the first phase and construction of the grid requires time proportional to n. It is also easy to see that the number of point-comparisons is a bound for the number of operations in the second phase. It is also possible to observe that the algorithm never compares a point pair if they are in non-adjacent cells, i.e., if their distance is at least $2d\sqrt{2}$.

This means we need to show that the expectation of $N(2d\sqrt{2})$ is proportional to n when d is chosen according to the algorithm in the first phase. Because of the observation from the last section, i.e., $N(2d\sqrt{2}) \leq 11^2 N(d)$, it is enough to verify that the expectation of $N(d)$ is linear.

Let us consider all pair distances: $d_1 := d(p_1, p_2)$, $d_2 := d(p_1, p_3)$, $\ldots$, $d_m := d(p_{n-1}, p_n)$ where $m = \frac{n(n-1)}{2}$.

Then we can find a permutation $\sigma : \{1, \ldots, m\} \to \{1, \ldots, m\}$, s.t., the distances are ordered, i.e., $d_{\sigma(i)} \leq d_{\sigma(j)}$ if $1 \leq i \leq j \leq m$.

The key observation is that $N(d_\sigma(i)) \leq i-1$, because N counts the number of point pairs which are closer than $d_{\sigma(i)}$, which can only be those corresponding to $d_{\sigma(1)}, d_{\sigma(2)}, \ldots, d_{\sigma(i-1)}$.

On the other hand the algorithm chooses the smallest of n random samples from $d_1, \ldots, d_m$. So the problem reduces to the computation of the expectation of the smallest element from n random samples from $1, \ldots, m$. The mean of this can be estimated to be $\frac{m+1}{n+1}$ which is in $\mathcal{O}(n)$. □

theory *Randomized-Closest-Pair-Time*
  imports
    Randomized-Closest-Pair-Growth
    Approximate-Model-Counting.ApproxMCAnalysis
    Distributed-Distinct-Elements.Distributed-Distinct-Elements-Balls-and-Bins
begin

lemma *time-sample-distance*: *map-pmf time* (*sample-distance ps*) = *return-pmf 1*
  unfolding *sample-distance-def time-bind-tpmf*
  by (*simp add*:*return-tpmf-def bind-return-pmf*) (*simp add*:*map-pmf-def*[*symmetric*] *time-lift-pmf*)

lemma *time-first-phase*:
  assumes *length ps* ≥ *2*
  shows *map-pmf time* (*first-phase ps*) = *return-pmf* (*2*∗*length ps*) (**is** *?L* = *?R*)
proof −
  let *?m* = *replicate-tpmf* (*length ps*) (*sample-distance ps*)

  have *ps-ne*: *ps* ≠ [] **using** *assms* **by** *auto*

have *?L* = *bind-pmf ?m* (λ*x*. *lift-tm* (*min-list-tm* (*val x*)) ≫= (λ*y*. *return-pmf* (*time x* + *time y*)))
  unfolding *first-phase-def time-bind-tpmf* **by** *simp*
also have ... = *bind-pmf ?m* (λ*x*. *return-pmf* (*time x* + *time* (*min-list-tm* (*val x*))))
  unfolding *lift-tm-def bind-return-pmf* **by** *simp*
also have ... = *bind-pmf ?m* (λ*x*. *return-pmf* (*time x* + *length* (*val x*)))
  using *ps-ne set-val-replicate-tpmf*(*1*) **by** (*intro bind-pmf-cong refl*
      arg-cong[**where** *f*=*return-pmf*] *arg-cong2*[**where** *f*=(+)] *time-min-list*) *fastforce*
also have ... = *bind-pmf ?m* (λ*x*. *return-pmf* (*time x* + *length ps*))
  using *set-val-replicate-tpmf*(*1*)
  by (*intro bind-pmf-cong refl arg-cong*[**where** *f*=*return-pmf*] *arg-cong2*[**where** *f*=(+)]) *auto*
also have ... = *map-pmf* (λ*x*. *x* + *length ps*) (*map-pmf time ?m*)
  unfolding *map-pmf-def*[*symmetric*] *map-pmf-comp* **by** *simp*
also have ... = *return-pmf* (*2* ∗ *length ps*)
  unfolding *time-replicate-tpmf time-sample-distance* **by** (*simp add*:*sum-list-replicate*)
finally show *?thesis* **by** *simp*
qed

lemma *time-build-grid*: *time* (*build-grid ps d*) = *length ps*
  unfolding *build-grid-def* **by** (*simp add*:*time-custom-tick*)

lemma *time-lookup-neighborhood*:
  time (*lookup-neighborhood* (*grid ps d*) *p*) ≤ *39*+*3*∗(*length*(*val*(*lookup-neighborhood* (*grid ps d*) *p*)))
  (**is** *?L* ≤ *?R*)
proof −
  define *s* **where** *s* = [(*0*, *0*), (*0*, *1*), (*1*, − *1*), (*1*, *0*), (*1*::*int*, *1*::*int*)]
  define *t* **where** *t* = *concat* (*map* (λ*x*. *filter* (λ*y*. *to-grid d y* = *x* + *to-grid d p*) *ps*) *s*)
  define *u* **where** *u* = *time* (*remove1-tm p t*)

  have *t-eq*: *length t*+*length s*=(∑*x*←*s*. *Suc* (*length* (*filter* (λ*y*. *to-grid d y*=*x*+*to-grid d p*) *ps*)))
    unfolding *t-def* **by** (*induction s*) *auto*

  have *a*:*u* ≤ *1* + *length t* **unfolding** *u-def* **using** *time-remove1* **by** *auto*

  have *?L* = *5*+*5*∗*length s* + *length t* + (*length t* + *length s*) + *u*
    unfolding *lookup-neighborhood-def s-def*[*symmetric*] *t-eq u-def*
    by (*simp add*:*time-map-tm comp-def grid-def sum-list-triv t-def*)
  also have ... = *5*+*6*∗*length s* + *2*∗*length t* + *u* **by** *simp*
  also have ... ≤ *5*+*6*∗*length s* + *2*∗*length t* + (*1*+*length t*) **using** *a* **by** *simp*
  also have ... = *36* + *3*∗*length t* **unfolding** *s-def* **by** *simp*
  also have ... ≤ *36* + *3* ∗ (*1*+*length* (*remove1 p t*))
    by (*intro add-mono mult-left-mono*) (*auto simp*:*length-remove1*)
  also have ... = *39* + *3*∗(*length* (*val* (*lookup-neighborhood* (*grid ps d*) *p*)))

```
    unfolding lookup-neighborhood-def s-def[symmetric] t-def
    by (simp add:val-remove1 comp-def grid-def)
  finally show ?thesis by simp
qed

lemma time-calc-dists-neighborhood:
  time (calc-dists-neighborhood (grid ps d) p) ≤
  40 + 5 ∗ (length (val (lookup-neighborhood (grid ps d) p))) (is ?L ≤ ?R)
proof −
  let ?g = grid ps d
 have ?L = 2∗ (length (val (lookup-neighborhood ?g p))) + 1 + time (lookup-neighborhood
?g p)
   unfolding calc-dists-neighborhood-def by (simp add:time-map-tm sum-list-triv)
  also have . . . ≤ 2∗ (length (val (lookup-neighborhood ?g p))) +1 +
    (39 + 3∗ (length (val (lookup-neighborhood ?g p))))
    by (intro add-mono mult-right-mono time-lookup-neighborhood) auto
  also have . . . = 40 + 5 ∗ (length (val (lookup-neighborhood ?g p))) by simp
  finally show ?thesis by simp
qed

lemma time-second-phase:
  fixes ps :: point list
  assumes d > 0 min-dist ps ≤ d length ps ≥ 2
  shows time (second-phase d ps) ≤ 2 + 44 ∗ length ps + 7 ∗ close-point-size ps
(2 ∗ sqrt 2 ∗ d)
    (is ?L ≤ ?R)
proof −
 define s where s = concat (map (λx. val (calc-dists-neighborhood (val (build-grid
ps d)) x)) ps)

  have len-s: length s = (∑x←ps. length (val (lookup-neighborhood (grid ps d)
x)))
   unfolding s-def by (simp add:calc-dists-neighborhood-def build-grid-val length-concat
comp-def)
  also have . . . = (∑x←ps. size (mset (val (lookup-neighborhood (grid ps d) x))))
    by simp
  also have . . . ≤
   (∑x←ps. size({#y∈# mset ps. to-grid d y−to-grid d x∈{(0,0),(0,1),(1,−1),(1,0),(1,1)}#}))
    unfolding lookup-neighborhood by (intro sum-list-mono size-mset-mono) simp
 also have . . . ≤ (∑x←ps. size({#y∈# mset ps. ∀k∈{1,2}. |⌊y$k/d⌋−⌊x$k/d⌋|≤1
#}))
   unfolding to-grid-def by (intro sum-list-mono size-mset-mono multiset-filter-mono-2)
auto
  also have . . . ≤ (∑x←ps. size({#y∈# mset ps. dist y x < d ∗ real-of-int (1 +
1) ∗ sqrt 2#}))
    using exhaust-2
   by (intro sum-list-mono size-mset-mono multiset-filter-mono-2 grid-dist-upperI[OF
assms(1)])
      blast
```

```
  also have ... = (∑ x←ps. length (filter (λy. dist x y < 2 * sqrt 2 * d) ps))
    by (simp add:dist-commute ac-simps) (metis mset-filter size-mset)
  also have ... = close-point-size ps ((2* sqrt 2)*d)
    unfolding close-point-size-def product-concat-map filter-concat length-concat
    by (simp add:comp-def)
  finally have len-s-bound: length s ≤ close-point-size ps (2* sqrt 2*d) by simp

  obtain u v where u ∈ set ps v ∈ set (val (lookup-neighborhood (grid ps d) u))
    using second-phase-aux[OF assms] that by metis
  hence False if length s = 0
    using that unfolding len-s sum-list-eq-0-iff by simp
  hence s-ne: s ≠ [] by auto

  have ?L = 2 + 4*length ps + (length s + time (min-list-tm s)) +
    (∑ i←ps. time (calc-dists-neighborhood (val (build-grid ps d)) i))
   unfolding second-phase-def by (simp add:time-map-tm s-def[symmetric] time-build-grid)
  also have ... ≤ 2 + 4*length ps + (length s + time (min-list-tm s)) +
    (∑ i←ps. 40+5* length (val (lookup-neighborhood (grid ps d) i)))
   unfolding build-grid-val by (intro add-mono sum-list-mono time-calc-dists-neighborhood)
auto
  also have ... = 2 + 44*length ps + (length s + time (min-list-tm s)) +
    (∑ i←ps. 5* length (val (lookup-neighborhood (grid ps d) i)))
    by (simp add:sum-list-addf sum-list-triv)
  also have ... = 2 + 44*length ps + 7*(length s)
    unfolding time-min-list[OF s-ne] len-s by (simp add:sum-list-const-mult)
  also have ... ≤ 2 + 44* length ps + 7 * close-point-size ps (2* sqrt 2*d)
    by (intro add-mono mult-left-mono len-s-bound) auto
  finally show  ?thesis by simp
qed

lemma mono-close-point-size: mono (close-point-size ps)
  unfolding close-point-size-def by (intro monoI length-filter-P-impl-Q) auto

lemma close-point-size-bound: close-point-size ps x ≤ length ps^2
 unfolding close-point-size-def power2-eq-square using length-filter-le length-product
by metis

lemma map-product: map (map-prod f g) (List.product xs ys) = List.product (map
f xs) (map g ys)
  unfolding product-concat-map by (simp add:map-concat comp-def)

lemma close-point-size-bound-2:
  close-point-size ps d ≤ length ps + 2 * card {(u,v). dist (ps!u) (ps!v)<d ∧ u<v
∧ v<length ps}
  (is ?L ≤ ?R)
proof -
  let ?n = length ps
  let ?h = λx. dist (ps ! fst x) (ps ! snd x) < d
   have e : List.product ps ps = map (map-prod ((!)ps) ((!) ps)) (List.product
```

```
[0..<?n] [0..<?n])
    unfolding map-product by (simp add:map-nth)

  have ?L = length (filter (λx. dist (ps ! fst x) (ps ! snd x) < d) (List.product[0..<?n][0..<?n]))
    unfolding close-point-size-def e by (simp add:comp-def case-prod-beta′)
  also have ... = card {x. ?h x ∧ fst x < ?n ∧ snd x < ?n}
   by (subst distinct-length-filter) (simp-all add:distinct-product Int-def mem-Times-iff)
  also have ... = card ({x. ?h x∧fst x<?n∧snd x<?n∧fst x ≠ snd x}∪{x. ?h
x∧fst x=snd x∧snd x<?n})
    by (intro arg-cong[where f=card]) auto
  also have ... ≤ card{x. ?h x∧fst x<?n∧snd x<?n∧fst x ≠ snd x}+card{x. ?h
x∧fst x=snd x∧snd x<?n}
    by (intro card-Un-le)
  also have ... ≤ card{x. ?h x∧fst x<?n∧snd x<?n∧fst x ≠ snd x}+card((λx.
(x,x))`{k. k<?n})
    by (intro add-mono order.refl card-mono finite-imageI) auto
  also have ... ≤ card{x. ?h x∧fst x<?n∧snd x<?n∧fst x ≠ snd x}+?n
    by (subst card-image) (auto intro:inj-onI)
  also have ... =card ({x. ?h x∧fst x<snd x∧snd x<?n}∪{x. ?h x∧snd x<fst
x∧fst x<?n})+?n
    by (intro arg-cong2[where f=(+)] arg-cong[where f=card]) auto
  also have ... ≤ (card {x. ?h x∧fst x<snd x∧snd x<?n} + card {x. ?h x∧snd
x<fst x∧fst x<?n})+?n
    by (intro add-mono card-Un-le order.refl)
  also have
      ...=(card{x. ?h x∧fst x<snd x∧snd x<?n}+card (prod.swap`{x. ?h x∧snd
x<fst x∧fst x<?n}))+?n
    by (subst card-image) auto
  also have ... = (card{x. ?h x∧fst x<snd x∧snd x<?n}+card ({x. ?h x∧fst
x<snd x∧snd x<?n}))+?n
   by (intro arg-cong2[where f=(+)] arg-cong[where f=card]) (auto simp:dist-commute)
  also have ... = ?R by (simp add:case-prod-beta′)
  finally show ?thesis by simp
qed

lemma card-card-estimate:
  fixes f :: ′a ⇒ (′b :: linorder)
  assumes finite S
  shows card {x ∈ S. a ≤ card {y ∈ S. f y < f x }} ≤ card S − a (is ?L ≤ ?R)
proof −
  define T where T = {x ∈ S. card {y ∈ S. f y < f x} < a}

  have T-range: T ⊆ S unfolding T-def by auto
  hence fin-T: finite T using assms finite-subset by auto

  have d:a ≤ card T ∨ T= S
  proof (rule ccontr)
    define x where x = arg-min-on f (S−T)
```

assume *a*:¬(*a* ≤ *card T* ∨ *T*=*S*)
hence *c*:*S* − *T* ≠ {} **using** *T-range* **by** *auto*
hence *b*:*x* ∈ *S*−*T* **using** *assms* **unfolding** *x-def* **by** (*intro arg-min-if-finite*) *auto*

have *False* **if** *y* ∈ *S*−*T f y* < *f x* **for** *y*
using *arg-min-if-finite*[*OF - c*] *that assms* **unfolding** *x-def* **by** *auto*
hence *card* {*y* ∈ *S*. *f y* < *f x*} ≤ *card T* **by** (*intro card-mono fin-T*) *auto*
also have . . . < *a* **using** *a* **by** *simp*
finally have *card* {*y* ∈ *S*. *f y* < *f x*} < *a* **by** *simp*
thus *False* **using** *b* **unfolding** *T-def* **by** *simp*
qed
have *?L* = *card* (*S* − *T*) **unfolding** *T-def* **by** (*intro arg-cong*[**where** *f*=*card*]) *auto*
also have . . . = *card S* − *card T* **using** *fin-T T-range* **by** (*intro card-Diff-subset*) *auto*
also have . . . ≤ *card S* − *a* **using** *d* **by** *auto*
finally show *?thesis* **by** *simp*
qed

lemma *finite-map-pmf*:
assumes *finite* (*set-pmf S*)
shows *finite* (*set-pmf* (*map-pmf f S*))
using *assms* **by** *simp*

lemma *finite-replicate-pmf*:
assumes *finite* (*set-pmf S*)
shows *finite* (*set-pmf* (*replicate-pmf n S*))
using *assms* **unfolding** *set-replicate-pmf lists-eq-set*
by (*simp add*:*finite-lists-length-eq*)

lemma *power-sum-approx*: ($\sum$ *k*<*m*. (*real k*)^*n*) ≤ *m*^(*n*+*1*)/*real* (*n*+*1*)
proof (*induction m*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc m*)
have ($\sum$ *k*<*Suc m*. *real k* ^ *n*) = ($\sum$ *k*<*m*. *real k* ^ *n*) + *real m* ^ *n* **by** *simp*
also have ... ≤ *real m* ^ (*n*+*1*) / *real* (*n*+*1*) + *real m*^*n* **by** (*intro add-mono Suc order.refl*)
also have ... = (*real m*^(*n*+*1*)+(*real* (*m*+*1*)−*m*)∗*real* (*n*+*1*)∗*real m*^((*n*+*1*)−*1*)) / *real* (*n*+*1*)
by (*simp add*:*field-simps*)
also have . . . ≤ (*real m*^(*n*+*1*)+(*real* (*m*+*1*)^(*n*+*1*)−*real m*^(*n*+*1*))) / *real* (*n*+*1*)
by (*intro divide-right-mono add-mono order.refl power-diff-est-2*) *simp-all*
also have . . . = *real* (*Suc m*) ^ (*n* + *1*) / *real* (*n* + *1*) **by** *simp*
finally show *?case* **by** *simp*
qed

lemma *exp-close-point-size*:
assumes *length ps* ≥ *2*
shows (∫ *d. real* (*close-point-size ps d*) ∂(*map-pmf val* (*first-phase ps*))) ≤ *2*∗ *real* (*length ps*)
(**is** *?L* ≤ *?R*)
proof −
let *?n* = *length ps*
define *T* **where** *T* = {*i. fst i*<*snd i*∧*snd i*<*?n*}
let *?I* = {..<*?n*}
let *?dpmf* = *map-pmf* (λ*i. dist* (*ps*!*fst i*) (*ps*!*snd i*)) (*pmf-of-set T*)
let *?q* = *prod-pmf* {..<*?n*} (λ-. *?dpmf*)
let *?h* = λ*x. dist* (*ps* ! *fst x*) (*ps* ! *snd x*)
let *?cps* = λ*d. card* {(*u*,*v*). *dist* (*ps*!*u*) (*ps*!*v*)<*d* ∧ *u*<*v* ∧ *v*<*length ps*}
let *?m* = *?n* ∗ (*?n* − *1*) *div 2*

have *card-T*: *card T* = *?m*
proof −
have *2* ∗ *card T* = *2* ∗ *card* {(*x*,*y*) ∈ {..<*?n*}×{..<*?n*}. *x* < *y*}
unfolding *T-def* **by** (*intro arg-cong*[**where** *f*=*card*] *arg-cong2*[**where** *f*=(∗)]) *auto*
also have ... = *card* {..<*?n*} ∗ (*card* {..<*?n*}−*1*) **by** (*intro card-ordered-pairs*) *simp*
also have ... = *?n* ∗ (*?n*−*1*) **by** *simp*
finally have *2* ∗ *card T*= *?n* ∗ (*?n*−*1*) **by** *simp*
thus *?thesis* **by** *simp*
qed

have *2* ∗ *1* ≤ *?n* ∗ (*?n*−*1*) **using** *assms* **by** (*intro mult-mono*) *auto*
hence *card T* > *0* **unfolding** *card-T* **using** *assms* **by** (*intro div-2-gt-zero*) *simp*
hence *T-fin-ne*: *finite T T* ≠ {} **by** (*auto simp*: *card-ge-0-finite*)

have *x-neI*:*x* ≠ [] **if** *x* ∈ *set-pmf* (*replicate-pmf ?n ?dpmf*) **for** *x*
using *that assms* **by** (*auto simp*:*set-replicate-pmf*)

have *a*:*map-pmf val* (*first-phase ps*) = *map-pmf min-list* (*replicate-pmf ?n ?dpmf*)
unfolding *first-phase-def val-tpmf-simps val-replicate-tpmf val-sample-distance T-def*[*symmetric*] *map-pmf-def*[*symmetric*] **by** (*intro map-pmf-cong val-min-list x-neI*) *auto*

hence *b*: {*x. t* < *?cps x*} = {} **if** *t* ∉ {..<*?m*} **for** *t*
proof −
have *?cps x* ≤ *card T* **for** *x*
using *T-fin-ne*(*1*) **unfolding** *T-def* **by** (*intro card-mono*) *auto*
moreover have *card T* ≤ *t* **using** *that* **unfolding** *card-T* **by** (*simp add*:*not-less*)
ultimately have *?cps x* ≤ *t* **for** *x* **using** *order.trans* **by** *auto*
thus *?thesis* **using** *not-less* **by** *auto*
qed

have *d*: {*y. t*< *?cps* (*min-list* (*map y* [*0*..<*?n*]))} = {..<*?n*}→{*y. t* < *?cps y*}

```
(is ?L2=?R2) for t
  proof (rule Set.set-eqI)
    fix x
    have x ∈ ?L2 ⟷ (t < ?cps (min-list (map x [0..<?n]))) by simp
    also have ... ⟷ (t < ?cps (Min (x ‘ {0..<?n})))
      using assms by (subst min-list-Min) auto
    also have ... ⟷ (t < Min (?cps ‘ x ‘ {0..<?n}))
        using assms by (intro arg-cong2[where f=(<)] mono-Min-commute refl
finite-imageI monoI
          card-mono finite-subset[OF - T-fin-ne(1)]) (auto simp:T-def)
    also have ... ⟷ (∀ i∈{0..<?n}. t < ?cps (x i))
      using assms by (subst Min-gr-iff) auto
    also have ... ⟷ x ∈ ?R2 by auto
    finally show x ∈ ?L2 ⟷ x ∈ ?R2 by simp
  qed

 have c: measure (replicate-pmf ?n ?dpmf) {x. t<?cps(min-list x)}≤(real (?m−(t+1))/real
?m)^?n
    (is ?L1 ≤ ?R1) for t
  proof −
     have ?L1 = measure(replicate-pmf(length [0..<?n]) ?dpmf) {x. t < ?cps
(min-list x)}
      by simp
   also have ... = measure (map-pmf (λf. map f [0..<?n]) (prod-pmf(set[0..<?n])(λ-.?dpmf)))
      {x. t<?cps(min-list x)}
     by (intro arg-cong2[where f=λx. measure (measure-pmf x)] replicate-pmf-Pi-pmf)
auto
    also have ... = measure ?q {y. t < ?cps (min-list (map y [0..<?n]))}
      by (simp add:atLeast0LessThan)
    also have ... = measure (prod-pmf {..<?n} (λ-. ?dpmf)) ({..<?n} → {y. t <
?cps y})
      unfolding d by simp
    also have ... = measure ?dpmf {y. t < ?cps y}^?n
      by (subst measure-Pi-pmf-Pi) simp-all
    also have ... = measure ?dpmf {y. t+1 ≤ ?cps y}^?n
      by (intro measure-pmf-cong arg-cong2[where f=(λx y. x^y)] refl) auto
     also have ... ≤ measure (pmf-of-set T) {y. t+1≤ card {x ∈ T. ?h x<?h
y}}^?n
      unfolding T-def by (auto simp:case-prod-beta' conj-commute)
     also have ... = (real (card {y∈T. t+1≤ card {x ∈ T. ?h x<?h y}})/real
(card T))^?n
      unfolding measure-pmf-of-set[OF T-fin-ne(2,1)] Int-def by simp
    also have ... ≤ (real (card T − (t+1))/real (card T))^?n
        by (intro power-mono divide-right-mono of-nat-mono card-card-estimate
T-fin-ne) auto
    also have ... = (real (?m − (t+1))/real ?m)^?n
      unfolding card-T by auto
    finally show ?thesis by simp
  qed
```

have *ennreal ?L* = (∫ *ds. real* (*close-point-size ps* (*min-list ds*)) ∂*replicate-pmf ?n ?dpmf*)
unfolding *a* **by** *simp*
also have ... ≤ (∫ *ds. real* (*?n* + *2*∗*?cps* (*min-list ds*)) ∂*replicate-pmf ?n ?dpmf*)
using *T-fin-ne*
by (*intro integral-mono-AE ennreal-leI AE-pmfI close-point-size-bound-2 of-nat-mono integrable-measure-pmf-finite finite-replicate-pmf*) *auto*
also have ... = *ennreal ?n* + *2*∗*ennreal*(∫ *ds. real* (*?cps* (*min-list ds*)) ∂*replicate-pmf ?n ?dpmf*)
by (*simp add*:*ennreal-mult′ integrable-measure-pmf-finite finite-replicate-pmf T-fin-ne*)
also have ... = *ennreal ?n* + *2*∗∫ ⁺ *x. ennreal* (*real* (*?cps* (*min-list x*))) ∂*replicate-pmf ?n ?dpmf*
by (*intro arg-cong2*[**where** *f*=(+)] *arg-cong2*[**where** *f*=(∗)] *finite-replicate-pmf nn-integral-eq-integral*[*symmetric*] *integrable-measure-pmf-finite*) (*auto simp*:*T-fin-ne*)
also have ... = *ennreal ?n* +*2*∗∫ ⁺ *x. ennreal-of-enat* (*?cps* (*min-list x*)) ∂*replicate-pmf ?n ?dpmf*
by (*intro nn-integral-cong arg-cong2*[**where** *f*=(+)] *arg-cong2*[**where** *f*=(∗)] *refl*)
(*simp add*: *ennreal-of-nat-eq-real-of-nat*)
also have ... = *ennreal ?n* +*2*∗(∑ *t. emeasure* (*replicate-pmf ?n ?dpmf*) {*x. t*<*?cps* (*min-list x*)})
by (*subst nn-integral-enat-function*) *simp-all*
also have ... =*ennreal ?n*+*2*∗(∑ *t*<*?m. emeasure*(*replicate-pmf ?n ?dpmf*){*x. t*<*?cps* (*min-list x*)})
using *b* **by** (*intro arg-cong2*[**where** *f*=(+)] *arg-cong2*[**where** *f*=(∗)] *sum-inf-finite*) *auto*
also have ... =*ennreal ?n*+*2*∗*ennreal*(∑ *t*<*?m. measure*(*replicate-pmf ?n ?dpmf*){*x. t*<*?cps*(*min-list x*)})
unfolding *measure-pmf.emeasure-eq-measure* **by** *simp*
also have ... ≤ *ennreal ?n*+*2*∗*ennreal* (∑ *t*<*?m.* (*real* (*?m* − (*t*+*1*))/*real ?m*)^*?n*)
by (*intro add-mono order.refl iffD2*[*OF ennreal-mult-le-mult-iff*] *ennreal-leI sum-mono c*) *auto*
also have ... = *ennreal ?n*+*ennreal* (*2*∗(∑ *t*<*?m.* (*real* (*?m* − (*t*+*1*))^*?n*/*real ?m*^*?n*)))
using *ennreal-mult′* **by** (*auto simp*:*algebra-simps power-divide*)
also have ... = *ennreal* (*real ?n* + (*2*∗(∑ *t*<*?m.* (*real* (*?m* − (*t*+*1*))^*?n*/*real ?m*^*?n*))))
by (*intro ennreal-plus*[*symmetric*] *mult-nonneg-nonneg sum-nonneg*) *simp-all*
also have ... = *ennreal* (*real ?n* + (*2*∗(∑ *t*<*?m.* (*real* (*?m* − (*t*+*1*))^*?n*))/*real ?m*^*?n*))
by (*simp add*:*sum-divide-distrib*[*symmetric*])
also have ... = *ennreal* (*real ?n* + (*2*∗(∑ *t*<*?m.* (*real t*^*?n*))/*real ?m*^*?n*))
by (*intro arg-cong*[**where** *f*=*ennreal*] *arg-cong2*[**where** *f*=(+)] *arg-cong2*[**where** *f*=(∗)]
arg-cong2[**where** *f*=(/)] *refl sum.reindex-bij-betw bij-betwI*[**where** *g*=λ*x. ?m* − (*x*+*1*)])

auto
also have ... ≤ *ennreal* (*real ?n* + (*2* ∗ (*real ?m*^(*?n*+*1*)/*real* (*?n* +*1*)))/*real ?m*^*?n*)
by (*intro ennreal-leI add-mono divide-right-mono mult-left-mono power-sum-approx*) *auto*
also have ... = *ennreal* (*real ?n* + (*2* ∗ (*real ?m*^(*?n*+*1*)/*real ?m*^*?n*)/ *real* (*?n* +*1*)))
by *simp*
also have ... = *ennreal* (*real ?n* + ((*2* ∗ *?m*)/ *real* (*?n*+*1*))) **by** (*simp add*:*field-simps*)
also have ... = *ennreal* (*real ?n* + (*?n*∗(*?n*−*1*)/ *real* (*?n*+*1*)))
by (*metis even-mult-iff even-numeral even-two-times-div-two odd-two-times-div-two-nat*)
also have ... = *ennreal* ((*real ?n*∗(*real ?n*+*1*) +*real ?n* ∗ (*real ?n*−*real 1*)) / *real* (*?n*+*1*))
using *assms* **by** (*subst of-nat-diff*[*symmetric*]) (*auto simp*:*field-simps*)
also have ... = *ennreal* (*2*∗*real ?n* ∗ *real ?n* / *real* (*?n*+*1*))
using *assms* **by** (*simp add*:*field-simps*)
also have ... ≤ *ennreal* (*2*∗*real ?n* ∗ *real ?n* / *real ?n*)
using *assms* **by** (*intro ennreal-leI mult-right-mono divide-left-mono mult-pos-pos*) *auto*
also have ... = *ennreal* (*2*∗*real ?n*) **by** *simp*
finally have *ennreal ?L* ≤ *ennreal* (*2*∗*real ?n*) **by** *simp*
thus *?L* ≤ *2*∗*real ?n* **by** *simp*
qed

definition *time-closest-pair* :: *real* ⇒ *real*
where *time-closest-pair n* = *2* + *1740* ∗ *n*

Main results of this section:

theorem *time-closest-pair*:
assumes *length ps* ≥ *2*
shows (∫ *x*. *real* (*time x*) ∂*closest-pair ps*) ≤ *time-closest-pair* (*length ps*) (**is** *?L* ≤ *?R*)
proof −
let *?n* = *length ps*
let *?cps* = *close-point-size ps*
let *?p* = *map-pmf val* (*first-phase ps*)

have (*0*,*1*) ∈ {*i*. *fst i* < *snd i* ∧ *snd i* < *length ps*} **using** *assms* **by** *auto*
hence *a*:*finite* {*i*. *fst i* < *snd i* ∧ *snd i* < *length ps*} {*i*. *fst i*<*snd i* ∧ *snd i*<*length ps*} ≠ {}
using *fin-nat-pairs*[**where** *n*=*length ps*] **by** (*auto simp*:*case-prod-beta′*)

have *finite* (*set-pmf* (*map-pmf val* (*sample-distance ps*)))
unfolding *sample-distance-def val-tpmf-simps map-pmf-def*[*symmetric*] **using** *a*
by (*intro finite-map-pmf*) *auto*

hence *int*[*simp*]: *integrable* (*measure-pmf* (*map-pmf val* (*first-phase ps*))) *f* **for** *f* :: *real* ⇒ *real*

unfolding *first-phase-def val-tpmf-simps val-replicate-tpmf* **unfolding** *map-pmf-def*[*symmetric*]
by (*metis integrable-measure-pmf-finite finite-replicate-pmf finite-map-pmf*)

have *map-pmf time* (*closest-pair ps*) = *first-phase ps* ⤜
(λ*x*. *return-pmf* (*if val x* = *0 then* (*tick 0*) *else second-phase* (*val x*) *ps*) ⤜
(λ*y*. *return-pmf* (*time x* + *time y*)))
using *time-first-phase*[*OF assms*]
unfolding *closest-pair-def time-bind-tpmf lift-tm-def if-distrib if-distribR* **by** *simp*
also have ... = *map-pmf* (λ*x*. *time x* + (*if val x* = *0 then 1 else time* (*second-phase* (*val x*) *ps*)))
(*first-phase ps*)
unfolding *bind-return-pmf map-pmf-def* **by** (*simp cong*:*if-cong*)
also have ... = *map-pmf* (λ*x*. *2*∗*length ps* +
(*if val x* = *0 then 1 else time* (*second-phase* (*val x*) *ps*))) (*first-phase ps*)
using *time-first-phase*[*OF assms*] **unfolding** *map-pmf-eq-return-pmf-iff*
by (*intro map-pmf-cong refl arg-cong2*[**where** *f*=(+)]) *simp*
also have ... = *map-pmf* (λ*x*. *2*∗*length ps* + (*if x*=*0 then 1 else time* (*second-phase x ps*))) *?p*
unfolding *map-pmf-comp* **by** *simp*
finally have *a*:*map-pmf time* (*closest-pair ps*) =
map-pmf (λ*x*. *2*∗*length ps* + (*if x*=*0 then 1 else time* (*second-phase x ps*))) *?p*
by *simp*

have (∫ *x*. *real* (*time x*) ∂*closest-pair ps*) = (∫ *x*. *real x* ∂*map-pmf time* (*closest-pair ps*))
by *simp*
also have ... = (∫ *d*. *2* ∗ *real ?n* + (*if d*=*0 then 1 else time* (*second-phase d ps*)) ∂*?p*)
unfolding *a* **by** *simp*
also have ... ≤ (∫ *d*. *2* ∗ *real ?n* + (*if d*≤*0 then 1 else 2*+*44*∗*?n*+*7*∗*?cps* ((*2*∗ *sqrt 2*)∗*d*)) ∂*?p*)
using *first-phase*[*OF assms*] *min-dist-nonneg*[*OF assms*] *order.trans* **unfolding** *AE-measure-pmf-iff*
by (*intro integral-mono-AE int AE-pmfI of-nat-mono mono-intros*
time-second-phase[*OF - - assms*(*1*)] *refl dual-order.not-eq-order-implies-strict*) *auto*
also have ... = (∫ *d*. *2*∗*real ?n*+(*if d*≤*0 then 1 else 2*+*44*∗*real ?n*+*7*∗*real*(*?cps* ((*2*∗ *sqrt 2*)∗*d*))) ∂*?p*)
by (*intro integral-cong-AE*) *simp-all*
also have ... ≤ (∫ *d*. *2* ∗ *real ?n* +
(*if d*≤*0 then 1 else 2*+*44*∗*real ?n*+*7*∗((*2*∗ *sqrt 2* ∗ (*2*∗ *sqrt 2*)+*3*)^*2* ∗ *real* (*?cps d*))) ∂*?p*)
using *growth-lemma*[**where** *a*=*2*∗ *sqrt 2*]
by (*intro integral-mono-AE int AE-pmfI mono-intros mult-right-mono*) *auto*
also have ... ≤
(∫ *d*. *2* ∗ *real ?n* + (*2*+*44*∗*real ?n*+*7*∗((*2*∗ *sqrt 2* ∗ (*2*∗ *sqrt 2*)+*3*)^*2* ∗ *real* (*?cps d*))) ∂*?p*)
by (*intro integral-mono-AE int AE-pmfI mono-intros mult-right-mono*) *simp*

also have ... $= (\int d.\ (2+46*real\ ?n)+847\ *\ real\ (?cps\ d)\ \partial ?p)$ **by** (*simp add*: *algebra-simps*)
also have ... $= (\int d.\ 2+46*real\ ?n\ \partial ?p)+(\int d.\ 847*\ real\ (?cps\ d)\ \partial ?p)$
by (*intro Bochner-Integration.integral-add int*)
also have ... $= (2+46*real\ ?n)+847*(\int d.\ real\ (?cps\ d)\ \partial ?p)$
by (*intro arg-cong2*[**where** $f=(+)$]) *simp-all*
also have ... $\leq (2+46*real\ ?n)+847*(2\ *\ real\ ?n)$
by (*intro mono-intros mult-left-mono exp-close-point-size assms*) *simp*
also have ... $= 2+1740*\ real\ ?n$ **by** *simp*
finally show *?thesis* **unfolding** *time-closest-pair-def* **by** *simp*
qed

theorem *asymptotic-time-closest-pair*:
time-closest-pair $\in O(\lambda x.\ x)$
unfolding *time-closest-pair-def* **by** *simp*

end

References

[1] B. Banyassady and W. Mulzer. A simple analysis of rabins algorithm for finding closest pairs. In *European Workshop on Computational Geometry (EuroCG)*, 2007.

[2] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

[3] S. Fortune and J. Hopcroft. A note on rabin's nearest-neighbor algorithm. *Information Processing Letters*, 8(1):20–23, 1979.

[4] S. Khuller and Y. Matias. A simple randomized sieve algorithm for the closest-pair problem. *Information and Computation*, 118(1):34–37, 1995.

[5] R. Lipton. Rabin flips a coin. https://rjlipton.com/2009/03/01/rabin-flips-a-coin/, 2009. Accessed: 2024-08-31.

[6] M. O. Rabin. Probabilistic algorithms. In *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39, USA, 1976. Academic Press, Inc.

[7] M. Rau and T. Nipkow. Closest pair of points algorithms. *Archive of Formal Proofs*, January 2020. https://isa-afp.org/entries/Closest_Pair_Points.html, Formal proof development.

[8] M. Rau and T. Nipkow. Verification of closest pair of points algorithms. In N. Peltier and V. Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 341–357, Cham, 2020. Springer International Publishing.