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
  ⟨proof⟩

lemma time-min-list: xs ≠ [] ⟹ time (min-list-tm xs) = length xs
  ⟨proof⟩
```

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*
⟨*proof*⟩

lemma *time-remove1*: *time* (*remove1-tm x ys*) ≤ *1* + *length ys*
⟨*proof*⟩

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* ⟨*proof*⟩

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* :: $nat \Rightarrow 'a\ pmf \Rightarrow 'a\ tpmf$
  where $\textit{lift-pmf}\ k\ m = \textit{map-pmf}\ (\lambda x.\ \textit{custom-tick}\ k \gg= (\lambda\text{-}.\ \textit{return}\ x))\ m$

adhoc-overloading *Monad-Syntax.bind* $\rightleftharpoons$ *bind-tpmf*

lemma *val-bind-tpmf*:
  $\textit{map-pmf}\ \textit{val}\ (\textit{bind-tpmf}\ m\ f) = \textit{map-pmf}\ \textit{val}\ m \gg= (\lambda x.\ \textit{map-pmf}\ \textit{val}\ (f\ x))$
  (**is** $?L = ?R$)
⟨*proof*⟩

lemma *val-return-tpmf*:
  $\textit{map-pmf}\ \textit{val}\ (\textit{return-tpmf}\ x) = \textit{return-pmf}\ x$
  ⟨*proof*⟩

lemma *val-lift-tpmf*: $\textit{map-pmf}\ \textit{val}\ (\textit{lift-pmf}\ k\ x) = x$
  ⟨*proof*⟩

lemma *val-lift-tm*:
  $\textit{map-pmf}\ \textit{val}\ (\textit{lift-tm}\ x) = \textit{return-pmf}\ (\textit{val}\ x)$
  ⟨*proof*⟩

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

lemma *time-return-tpmf*: $\textit{map-pmf}\ \textit{time}\ (\textit{return-tpmf}\ x) = \textit{return-pmf}\ 0$
  ⟨*proof*⟩

lemma *time-lift-pmf*: $\textit{map-pmf}\ \textit{time}\ (\textit{lift-pmf}\ x\ p) = \textit{return-pmf}\ x$
  ⟨*proof*⟩

lemma *time-bind-tpmf*: $\textit{map-pmf}\ \textit{time}\ (\textit{bind-tpmf}\ m\ f) =$
  do {
    $x \leftarrow m$;
    $y \leftarrow f\ (\textit{val}\ x)$;
    $\textit{return-pmf}\ (\textit{time}\ x + \textit{time}\ y)$
  }
  ⟨*proof*⟩

lemma *bind-return-tm*: $\textit{bind-tm}\ (\textit{Time-Monad.return}\ x)\ f = f\ x$
  ⟨*proof*⟩

lemma *bind-return-tpmf*: $\textit{bind-tpmf}\ (\textit{return-tpmf}\ x)\ f = (f\ x)$
  ⟨*proof*⟩

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

fun *replicate-tpmf* :: $nat \Rightarrow 'a\ tpmf \Rightarrow 'a\ list\ tpmf$
  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*⟩

lemma *val-replicate-tpmf*:
map-pmf val (*replicate-tpmf n x*) = *replicate-pmf n* (*map-pmf val x*)
⟨*proof*⟩

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*⟩

lemma *replicate-return-pmf*[*simp*]: *replicate-pmf n* (*return-pmf x*) = *return-pmf* (*replicate n x*)
⟨*proof*⟩

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})
  ⟨proof⟩
```

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} ⟨proof⟩

lemma min-dist-set-fin: finite {dist x y|x y. {#x, y#} ⊆# mset xs}
⟨proof⟩

lemma min-dist-ne: length xs ≥ 2 ⟷ {dist x y|x y. {# x,y#} ⊆# mset xs} ≠
{} (is ?L ⟷ ?R)
⟨proof⟩
lemmas min-dist-neI = iffD1[OF min-dist-ne]

lemma min-dist-nonneg:
  assumes length xs ≥ 2
```

shows *min-dist xs* $\geq$ *0*
⟨*proof*⟩

lemma *min-dist-pos-iff*:
assumes *length xs* $\geq$ *2*
shows *distinct xs* $\longleftrightarrow$ *0* $<$ *min-dist xs*
⟨*proof*⟩

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*⟩

lemma *filter-mset-disj*:
filter-mset ($\lambda x.\ p\ x \vee q\ x$) *xs* = *filter-mset* ($\lambda x.\ p\ x \wedge \neg\ q\ x$) *xs* + *filter-mset q xs*
⟨*proof*⟩

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

lemma *size-filter-mset-decompose'*:
size (*filter-mset* ($\lambda x.\ f\ x \in T$) *xs*) = *sum'* (λt. *size* (*filter-mset* ($\lambda x.\ f\ x = t$) *xs*)) *T*
(**is** *?L* = *?R*)
⟨*proof*⟩

lemma *filter-product*:
filter ($\lambda x.\ P$ (*fst x*)$\wedge Q$ (*snd x*)) (*List.product xs ys*) = *List.product* (*filter P xs*) (*filter Q ys*)
⟨*proof*⟩

lemma *floor-diff-bound*: $|\lfloor x\rfloor - \lfloor y\rfloor| \leq \lceil |x - (y::real)| \rceil$ ⟨*proof*⟩

lemma *power2-strict-mono*:
fixes *x y* :: *'a* :: *linordered-idom*
assumes $|x| < |y|$
shows *x^2* < *y^2*
⟨*proof*⟩

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

lemma *build-grid-val*: *val* (*build-grid ps d*) = *grid ps d*
⟨*proof*⟩

lemma *lookup-neighborhood*:
  $mset\ (val\ (lookup\text{-}neighborhood\ (grid\ ps\ d)\ p)) =$
  $filter\text{-}mset\ (\lambda x.\ to\text{-}grid\ d\ x\ -\ to\text{-}grid\ d\ p \in \{(0,0),(0,1),(1,-1),(1,0),(1,1)\})$
$(mset\ ps) - \{\#p\#\}$
⟨*proof*⟩

lemma *fin-nat-pairs*: $finite\ \{(i,\ j).\ i < j \land j < (n::nat)\}$
  ⟨*proof*⟩

lemma *mset-list-subset*:
  assumes $distinct\ ys\ set\ ys \subseteq \{..<length\ xs\}$
  shows $mset\ (map\ ((!)\ xs)\ ys) \subseteq\#\ mset\ xs$ (**is** $?L \subseteq\#\ ?R$)
⟨*proof*⟩

lemma *sample-distance*:
  assumes $length\ ps \geq 2$
  shows $AE\ d\ in\ map\text{-}pmf\ val\ (sample\text{-}distance\ ps).\ min\text{-}dist\ ps \leq d$
⟨*proof*⟩

lemma *first-phase*:
  assumes $length\ ps \geq 2$
  shows $AE\ d\ in\ map\text{-}pmf\ val\ (first\text{-}phase\ ps).\ min\text{-}dist\ ps \leq d$
⟨*proof*⟩

definition $grid\text{-}lex\text{-}ord :: int * int \Rightarrow int * int \Rightarrow bool$
  where $grid\text{-}lex\text{-}ord\ x\ y = (fst\ x < fst\ y \lor (fst\ x = fst\ y \land snd\ x \leq snd\ y))$

lemma *grid-lex-order-antisym*: $grid\text{-}lex\text{-}ord\ x\ y \lor grid\text{-}lex\text{-}ord\ y\ x$
  ⟨*proof*⟩

lemma *grid-dist*:
  fixes $p\ q :: point$
  assumes $d > 0$
  shows $|\lfloor p\ \$\ k/d\rfloor - \lfloor q\ \$\ k/d\rfloor| \leq \lceil dist\ p\ q/d\rceil$
⟨*proof*⟩

lemma *grid-dist-2*:
  fixes $p\ q :: point$
  assumes $d > 0$
  assumes $\lceil dist\ p\ q/d\rceil \leq s$
  shows $to\text{-}grid\ d\ p\ -\ to\text{-}grid\ d\ q \in \{-s..s\}\times\{-s..s\}$
⟨*proof*⟩

lemma *grid-dist-3*:
  fixes $p\ q :: point$
  assumes $d > 0$
  assumes $\lceil dist\ q\ p/d\rceil \leq 1\ grid\text{-}lex\text{-}ord\ (to\text{-}grid\ d\ p)\ (to\text{-}grid\ d\ q)$
  shows $to\text{-}grid\ d\ q\ -\ to\text{-}grid\ d\ p \in \{(0,0),(0,1),(1,-1),(1,0),(1,1)\}$
⟨*proof*⟩

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*⟩

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*⟩

Main result of this section:

theorem *closest-pair-correct*:
  assumes *length ps ≥ 2*
  shows *AE r in map-pmf val (closest-pair ps). r = min-dist ps*
⟨*proof*⟩

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*⟩

lemma *of-nat-sum′*:
  (*of-nat* (*sum′ f S*) :: (′*a* :: {*semiring-char-0*})) = *sum′* (λ*x*. *of-nat* (*f x*)) *S*
  ⟨*proof*⟩

lemma *sum′-nonneg*:
  fixes *f* :: ′*a* ⇒ ′*b* :: {*ordered-comm-monoid-add*}

assumes $\bigwedge x.\ x \in S \Longrightarrow f\ x \geq 0$
shows $sum'\ f\ S \geq 0$
$\langle proof \rangle$

lemma *sum'-mono*:
fixes $f :: 'a \Rightarrow 'b :: \{ordered\text{-}comm\text{-}monoid\text{-}add\}$
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** $?L \leq ?R$)
$\langle proof \rangle$

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** $?L \leq ?R$)
$\langle proof \rangle$

context *comm-monoid-set*
begin

lemma *reindex-bij-betw'*:
assumes $bij\text{-}betw\ h\ S\ T$
shows $G\ (\lambda x.\ g\ (h\ x))\ S = G\ g\ T$
$\langle proof \rangle$

end

definition $close\text{-}point\text{-}size\ xs\ d = length\ (filter\ (\lambda(p,q).\ dist\ p\ q < d)\ (List.product\ xs\ xs))$

lemma *grid-dist-upper*:
fixes $p\ q :: point$
assumes $d > 0$
shows $dist\ p\ q < sqrt\ (\sum i{\in}UNIV.(d*(|\lfloor p\$i/d\rfloor - \lfloor q\$i/d\rfloor|+1))\hat{}2)$
(**is** $?L < ?R$)
$\langle proof \rangle$

lemma *grid-dist-upperI*:
fixes $p\ q :: point$
fixes $d :: real$
assumes $d > 0$
assumes $\bigwedge k.\ |\lfloor p\$k/d\rfloor - \lfloor q\$k/d\rfloor| \leq s$
shows $dist\ p\ q < d * (s+1) * sqrt\ 2$
$\langle proof \rangle$

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* ≡ (*λx*. *real* (*length* (*filter* (*λp*. *to-grid e p* = *x*) *xs*)))
shows *close-point-size xs d* ≤ ($\sum$ *i* ∈ {−*s*..*s*}×{−*s*..*s*}. *sum′* (*λx*. *G x* ∗ *G* (*x*+*i*)) *UNIV*)
(**is** *?L* ≤ *?R*)
⟨*proof*⟩

lemma *close-point-approx-lower*:
fixes *xs* :: *point list*
fixes *G* :: *int* × *int* ⇒ *real*
fixes *d* :: *real*
assumes *d* > *0*
defines *G* ≡ (*λx*. *real* (*length* (*filter* (*λp*. *to-grid d p* = *x*) *xs*)))
shows *sum′* (*λx*. *G x* ^ *2*) *UNIV* ≤ *close-point-size xs* (*d* ∗ *sqrt 2*)
(**is** *?L* ≤ *?R*)
⟨*proof*⟩

lemma *build-grid-finite*:
assumes *inj f*
shows *finite* {*x*. *filter* (*λp*. *to-grid d p* = *f x*) *xs* ≠ []}
⟨*proof*⟩

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*⟩

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: $d_1 := d(p_1, p_2)$, $d_2 := d(p_1, p_3)$, ..., $d_m := d(p_{n-1}, p_n)$ 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*
  ⟨*proof*⟩

lemma *time-first-phase*:
  assumes *length ps* ≥ *2*
  shows *map-pmf time* (*first-phase ps*) = *return-pmf* (*2*∗*length ps*) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *time-build-grid*: *time* (*build-grid ps d*) = *length ps*
  ⟨*proof*⟩

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*⟩

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*⟩

lemma *time-second-phase*:
  fixes *ps* :: *point list*

assumes $d > 0$ *min-dist ps* $\leq$ *d length ps* $\geq$ *2*
shows *time* (*second-phase d ps*) $\leq$ *2* + *44* $*$ *length ps* + *7* $*$ *close-point-size ps* (*2* $*$ *sqrt 2* $*$ *d*)
(**is** *?L* $\leq$ *?R*)
⟨*proof*⟩

lemma *mono-close-point-size*: *mono* (*close-point-size ps*)
⟨*proof*⟩

lemma *close-point-size-bound*: *close-point-size ps x* $\leq$ *length ps*^*2*
⟨*proof*⟩

lemma *map-product*: *map* (*map-prod f g*) (*List.product xs ys*) = *List.product* (*map f xs*) (*map g ys*)
⟨*proof*⟩

lemma *close-point-size-bound-2*:
close-point-size ps d $\leq$ *length ps* + *2* $*$ *card* {(*u*,*v*). *dist* (*ps*!*u*) (*ps*!*v*)<*d* $\wedge$ *u*<*v* $\wedge$ *v*<*length ps*}
(**is** *?L* $\leq$ *?R*)
⟨*proof*⟩

lemma *card-card-estimate*:
fixes *f* :: *'a* $\Rightarrow$ (*'b* :: *linorder*)
assumes *finite S*
shows *card* {*x* $\in$ *S*. *a* $\leq$ *card* {*y* $\in$ *S*. *f y* < *f x* }} $\leq$ *card S* − *a* (**is** *?L* $\leq$ *?R*)
⟨*proof*⟩

lemma *finite-map-pmf*:
assumes *finite* (*set-pmf S*)
shows *finite* (*set-pmf* (*map-pmf f S*))
⟨*proof*⟩

lemma *finite-replicate-pmf*:
assumes *finite* (*set-pmf S*)
shows *finite* (*set-pmf* (*replicate-pmf n S*))
⟨*proof*⟩

lemma *power-sum-approx*: ($\sum$ *k*<*m*. (*real k*)^*n*) $\leq$ *m*^(*n*+*1*)/*real* (*n*+*1*)
⟨*proof*⟩

lemma *exp-close-point-size*:
assumes *length ps* $\geq$ *2*
shows ($\int$ *d*. *real* (*close-point-size ps d*) ∂(*map-pmf val* (*first-phase ps*))) $\leq$ *2*$*$ *real* (*length ps*)
(**is** *?L* $\leq$ *?R*)
⟨*proof*⟩

definition *time-closest-pair* :: *real* $\Rightarrow$ *real*

where *time-closest-pair* $n = 2 + 1740 * n$

Main results of this section:

theorem *time-closest-pair*:
assumes *length ps* ≥ 2
shows $(\int x.\ real\ (time\ x)\ \partial closest\text{-}pair\ ps) \leq time\text{-}closest\text{-}pair\ (length\ ps)$ (**is** $?L \leq ?R$)
$\langle proof \rangle$

theorem *asymptotic-time-closest-pair*:
$time\text{-}closest\text{-}pair \in O(\lambda x.\ x)$
$\langle proof \rangle$

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.