Illuminating next generation sequencing data with Go

Dan Kortschak
Bioinformatics Group
School of Molecular and Biomedical Science
The University of Adelaide

Outline

- The Problem
- Background
- Approach
- Implementation
- Summary

The Problem

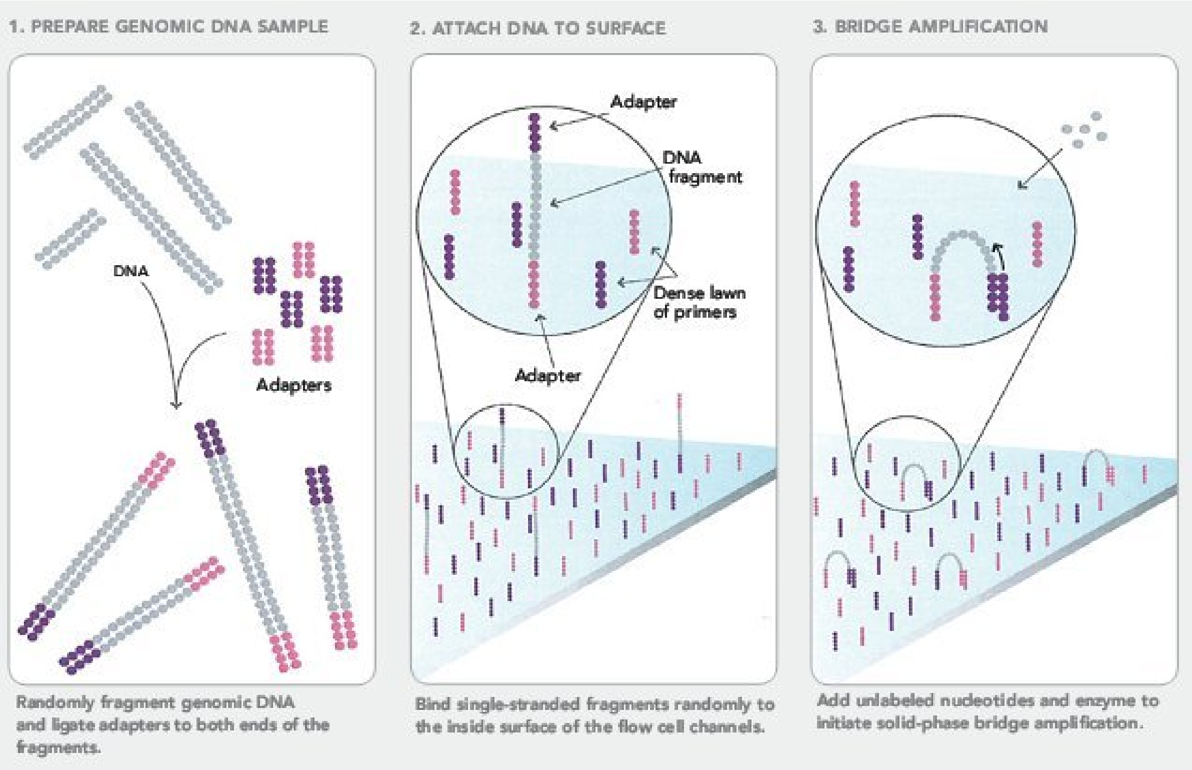
Examining discordant read pairs for Structural Variant detection

How often does a discordant pair arise due to the sequencing technology as opposed to the underlying biology?

Background

Illumina sequencing technology (briefly)

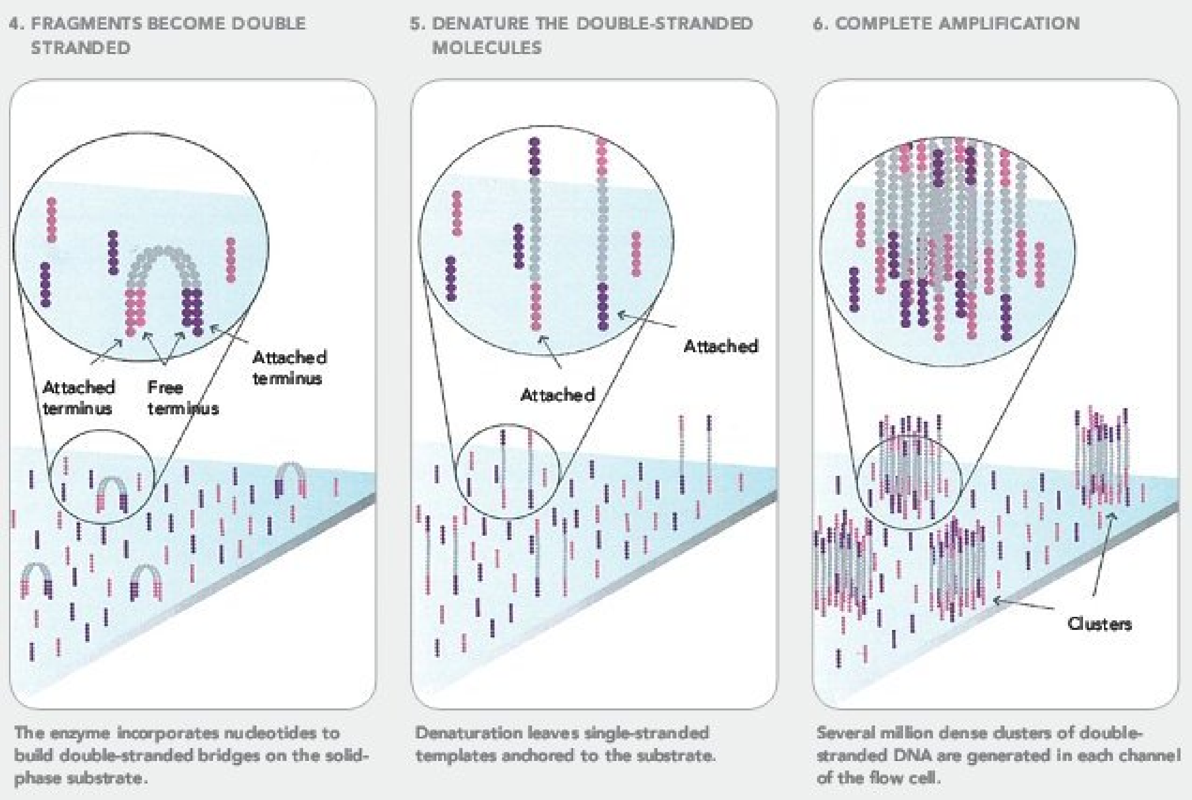
Prepare fragments, bind to substrate and bridge.


1. PREPARE GENOMIC DNA SAMPLE
DNA
Adapters
Randomly fragment genomic DNA and ligate adapters to both ends of the fragments.
2. ATTACH DNA TO SURFACE
Adapter
DNA fragment
Dense lawn of primers
Adapter
Bind single-stranded fragments randomly to the inside surface of the flow cell channels.
3. BRIDGE AMPLIFICATION
Add unlabeled nucleotides and enzyme to initiate solid-phase bridge amplification.

```
Illumina 2007
```

Illumina tech

Form 'polonies' of amplified DNA.



4. FRAGMENTS BECOME DOUBLE STRANDED
Attached terminus
Free terminus
Attached terminus
5. DENATURE THE DOUBLE-STRANDED MOLECULES
Attached
Attached
6. COMPLETE AMPLIFICATION
Clusters

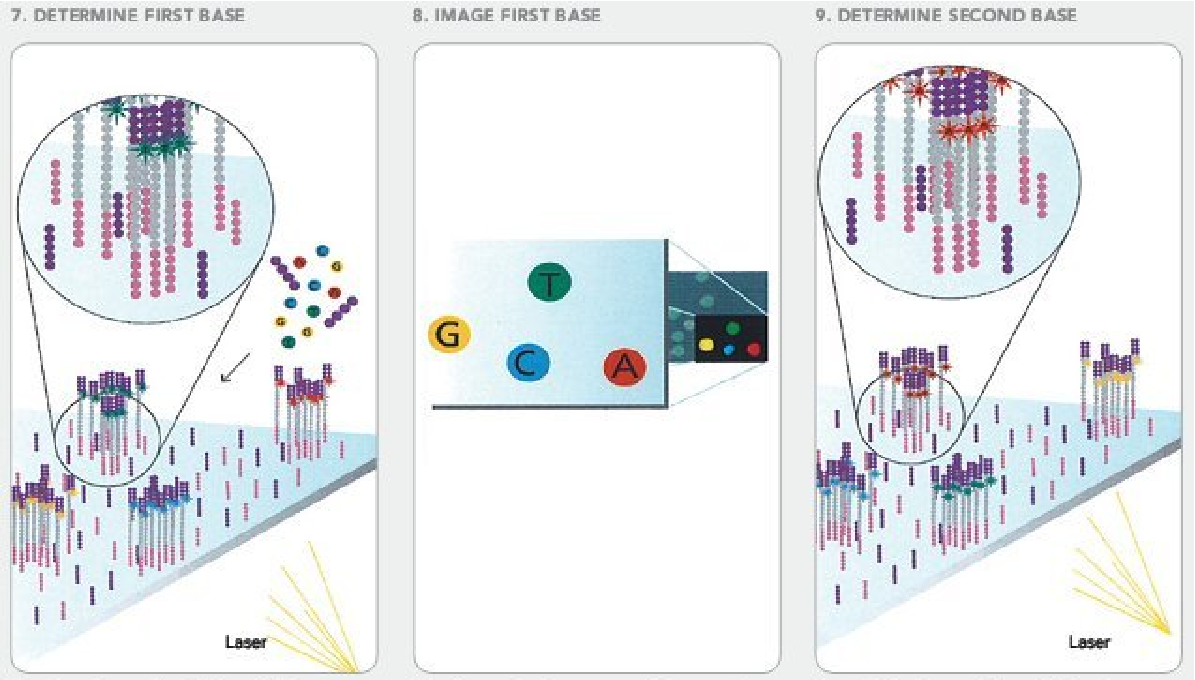
The enzyme incorporates nucleotides to build double-stranded bridges on the solid-phase substrate.

Denaturation leaves single-stranded templates anchored to the substrate.

Several million dense clusters of double-stranded DNA are generated in each channel of the flow cell.

Illumina tech

Perform step-wise polymer extension - detection by fluorescence.



7. DETERMINE FIRST BASE

First chemistry cycle: to initiate the first sequencing cycle, add all four labeled reversible terminators, primers and DNA polymerase enzyme to the flow cell.

8. IMAGE FIRST BASE

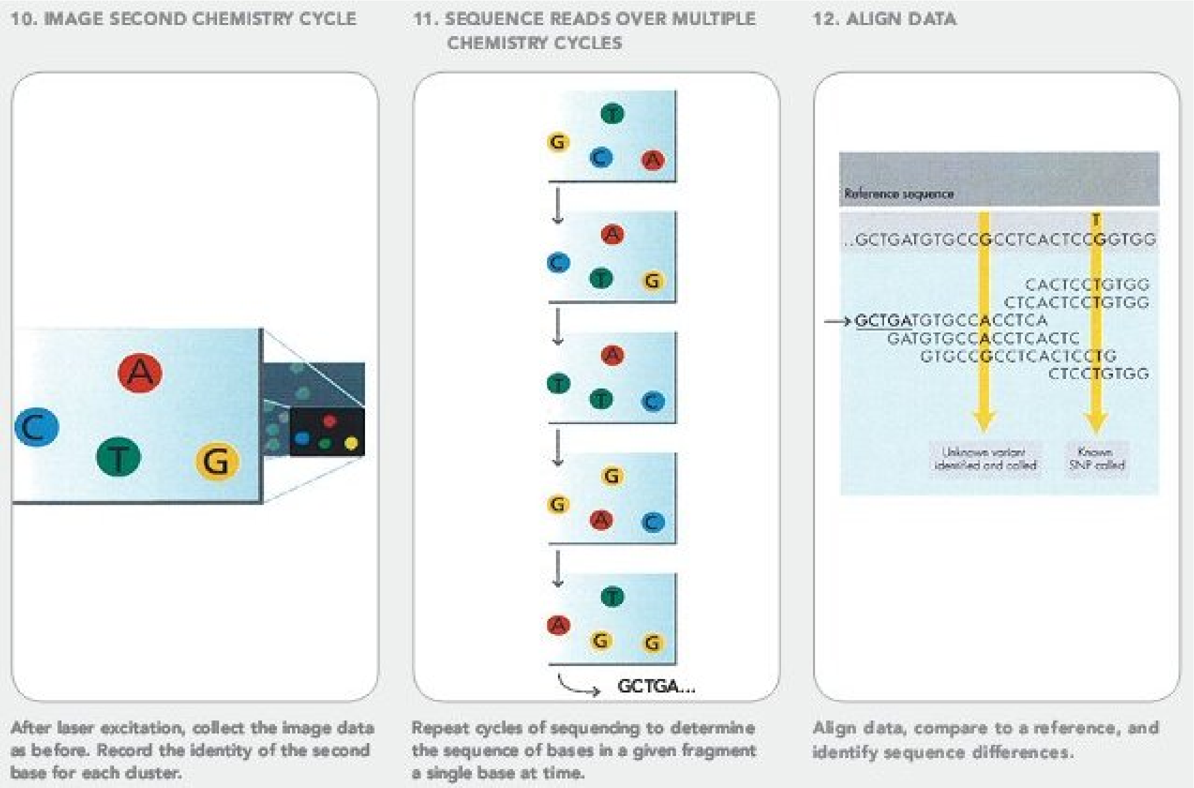
After laser excitation, capture the image of emitted fluorescence from each cluster on the flow cell. Record the identity of the first base for each cluster.

9. DETERMINE SECOND BASE

Second chemistry cycle: to initiate the next sequencing cycle, add all four labeled reversible terminators and enzyme to the flow cell.

Illumina tech

Image processing and sequence generation.



10. IMAGE SECOND CHEMISTRY CYCLE

11. SEQUENCE READS OVER MULTIPLE CHEMISTRY CYCLES

12. ALIGN DATA

After laser excitation, collect the image data as before. Record the identity of the second base for each cluster.

Repeat cycles of sequencing to determine the sequence of bases in a given fragment a single base at time.

Align data, compare to a reference, and identify sequence differences.

Approach

How often are discordant pairs explainable by polony coincidence?

Algorithm based on simplifying assumption: only one polony explains a discordant pair, that polony is a concordant pair.

- Save memory/time and make code simpler — probably unjustified

```
Type reads:
    collection of mapped read pairs, including flow cell metadata.
```

```
Function Collision(R reads):
    Let S be a spatially indexed store of reads
    Let C and D be 0
    For each read pair r in R:
        Increment D
        If is-discordant(r):
            Store r in S
    For each read pair r in R:
        If is-not-discordant(r):
            Let c be closest read to r in S
            If mapping-of(c) overlaps mapping-of(r)
                Increment C
    Return C, D.
```

Implementation

Reading the data

```
// Package boom is a wrapper for the samtools bam library.
```

```
func main() {
    bf, err := boom.OpenBAM("sample.bam")
    if err != nil {
        fmt.Fprintf(os.Stderr, "could not open file: %v\n", err)
        os.Exit(1)
    }
    fmt.Println(bf.RefNames())

    for {
        r, _, err := bf.Read()
        if err != nil {
            break
        }
        fmt.Println(r)
    }
}
```

Run

There is a pure Go package, but it is not yet mature.

- Not feature complete and only single threaded

Retrieving the spatial data from a read

```
// Package illumina provides support for handling Illumina read metadata.
```

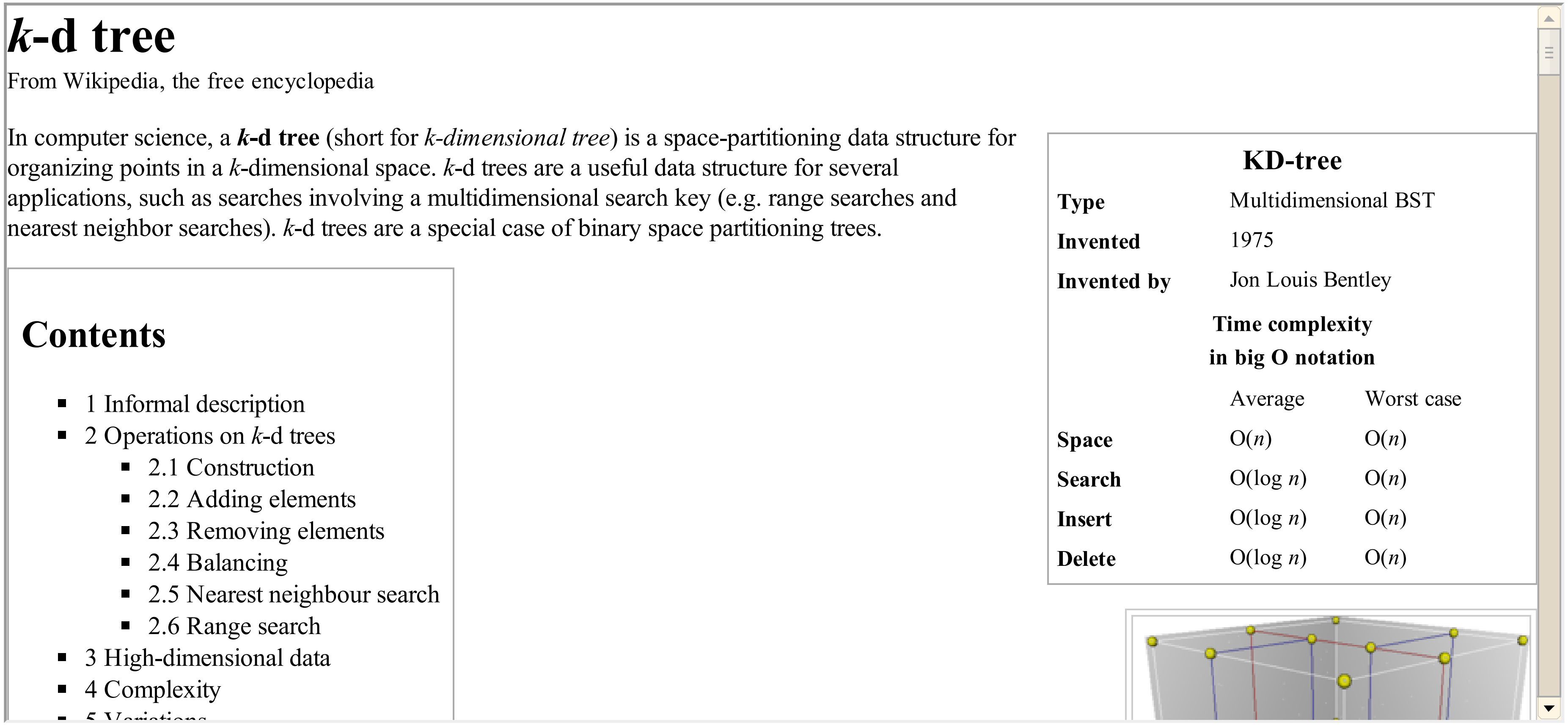
```
func main() {
    reads := []Read{
        {"HWUSI-EAS100R:6:73:941:1973#ATCACG/1", ""},
        {"EAS139:136:FC706VJ:2:2104:15343:197393", "1:Y:18:ATCACG"},
    }

    for _, r := range reads {
        m, err := illumina.Parse(r)
        if err != nil {
            fmt.Println(err)
        } else {
            spew.Dump(m)
        }
    }
}
```

Run

A spatial store

```
    Let S be a spatially indexed store of reads
```



k-d tree

From Wikipedia, the free encyclopedia

In computer science, a ***k*-d tree** (short for *k-dimensional tree*) is a space-partitioning data structure for organizing points in a *k*-dimensional space. *k*-d trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches). *k*-d trees are a special case of binary space partitioning trees.

KD-tree		
Type	Multidimensional BST	
Invented	1975	
Invented by	Jon Louis Bentley	
	Time complexity in big O notation	
	Average	Worst case
Space	O(*n*)	O(*n*)
Search	O(log *n*)	O(*n*)
Insert	O(log *n*)	O(*n*)
Delete	O(log *n*)	O(*n*)

Contents

There are others.

k-d tree

```
// Package kdtree implements a k-d tree.
```

```
package main

import (
    "code.google.com/p/biogo.kdtree"

    "fmt"
    "math"
)

var wpData = kdtree.Points{{2, 3}, {5, 4}, {9, 6}, {4, 7}, {8, 1}, {7, 2}}

func main() {
    t := kdtree.New(wpData, false)
    q := kdtree.Point{8, 7}
    p, d := t.Nearest(q)
    fmt.Printf("%v is closest point to %v, d=%f\n", p, q, math.Sqrt(d))
}
```

Run

Interfaces (interlude)

Go makes extensive use of structural typing. From the kdtree package...

```
// A Comparable is the element interface for values stored in a k-d tree.
type Comparable interface {
    // Compare returns the shortest translation of the plane through b with
    // normal vector along dimension d to the parallel plane through a.
    //
    // Given c = a.Compare(b, d):
    //  c = a_d - b_d
    //
    Compare(Comparable, Dim) float64

    // Dims returns the number of dimensions described in the Comparable.
    Dims() int

    // Distance returns the squared Euclidian distance between the receiver and
    // the parameter.
    Distance(Comparable) float64
}
```

The `kdtree.Point` type satisfies this interface, but we can make our own.

Implement a type that satisfies kdtree.Comparable

```
type illuminaRecord struct {
    A, B mapping
    illumina.Metadata
}
```

```
func (p *illuminaRecord) Compare(c kdtree.Comparable, d kdtree.Dim) float64 {
    q := c.(*illuminaRecord)
    switch d {
    case 0:
        return float64(p.Coordinate.X-q.Coordinate.X) * xunit
    case 1:
        return float64(p.Coordinate.Y-q.Coordinate.Y) * yunit
    default:
        panic("illegal dimension")
    }
}
func (p *illuminaRecord) Dims() int { return 2 }
func (p *illuminaRecord) Distance(c kdtree.Comparable) float64 {
    q := c.(*illuminaRecord)
    x := float64(p.Coordinate.X-q.Coordinate.X) * xunit
    y := float64(p.Coordinate.Y-q.Coordinate.Y) * yunit
    return x*x + y*y
}
```

k-d tree performance is sensitive to input order

Balanced trees perform better. This requires identification of the median points.

The kdtree package allows you to do this...

```
type Interface interface {
    // Index returns the ith element of the list of points.
    Index(i int) Comparable

    // Len returns the length of the list.
    Len() int

    // Pivot partitions the list based on the dimension specified.
    Pivot(Dim) int

    // Slice returns a slice of the list.
    Slice(start, end int) Interface
}
```

The `kdtree.Points` type satisfied this interface, and again we can make our own.

Implement a type that satisfies kdtree.Interface

Using this type, we can construct a close to optimal tree with the `kdtree.`New function.

```
// New returns a k-d tree constructed from the values in p.
func New(p Interface, bounding bool) *Tree
```

We want a slice of illumina records that can determine the median for each dimension.

```
type illuminaRecords []*illuminaRecord

func (p illuminaRecords) Index(i int) kdtree.Comparable { return p[i] }
func (p illuminaRecords) Len() int                      { return len(p) }
func (p illuminaRecords) Pivot(d kdtree.Dim) int {
    return plane{illuminaRecords: p, Dim: d}.Pivot()
}
func (p illuminaRecords) Slice(start, end int) kdtree.Interface { return p[start:end] }
```

But, you can see `Pivot` depends on another type, `plane`, to allow the pivot to be performed with respect to a specific dimension.

The plane helper type - sort based on a dimension

```
type plane struct {
    kdtree.Dim
    illuminaRecords
}

func (p plane) Less(i, j int) bool {
    switch p.Dim {
    case 0:
        return p.illuminaRecords[i].Coordinate.X < p.illuminaRecords[j].Coordinate.X
    case 1:
        return p.illuminaRecords[i].Coordinate.Y < p.illuminaRecords[j].Coordinate.Y
    default:
        panic("illegal dimension")
    }
}
func (p plane) Pivot() int { return kdtree.Partition(p, kdtree.MedianOfRandoms(p, randoms)) }
func (p plane) Slice(start, end int) kdtree.SortSlicer {
    p.illuminaRecords = p.illuminaRecords[start:end]
    return p
}
func (p plane) Swap(i, j int) {
    p.illuminaRecords[i], p.illuminaRecords[j] = p.illuminaRecords[j], p.illuminaRecords[i]
}
```

Reality is messy

We need to consider that polony addresses are more complicated than just an x, y-coordinate pair.

- Flow cell
- Lane
- Tile

To avoid collisions between coordinates in different spaces, we keep a collection of illuminaRecords with a look up table based on these values:

```
type tileAddress struct {
    FlowCell string
    Lane     int8
    Tile     int
}
```

```
meta := make(map[tileAddress]illuminaRecords)
```

Collecting the read data

Bundle up all the relevant read information.

```
func newRecord(r [2]*boom.Record, names []string) (*illuminaRecord, error) {
    m, err := illumina.Parse(boomIllumina{r[0]}) // They are a pair, so we only parse one.
    if err != nil {
        return nil, err
    }
    return &illuminaRecord{
        A: mapping{
            Segment: names[r[0].RefID()],
            Start:   r[0].Start(),
            End:     r[0].End(),
        },
        B: mapping{
            Segment: names[r[1].RefID()],
            Start:   r[1].Start(),
            End:     r[1].End(),
        },
        Metadata: m,
    }, nil
}
```

Building the data sets for storage

Add the record to the relevant collection.

```
const (
    filterReq  = 0
    filterMask = boom.Unmapped | boom.MateUnmapped | boom.Secondary | boom.Duplicate |
                 boom.ProperPair | filterReq
)

if r[0].Flags()&filterMask == filterReq && r[1].Flags()&filterMask == filterReq {
    discordant++
    m, err := newRecord(r, names)
    if err != nil {
        panic(err)
    }

    ta := tileAddress{
        FlowCell: m.FlowCell,
        Lane:     m.Lane,
        Tile:     m.Tile,
    }
    meta[ta] = append(meta[ta], m)
}
```

Construct the trees

Loop over the collections and create the stores for searching.

Create a collection of trees.

```
    ts := make(map[tileAddress]*kdtree.Tree)
```

Construct each tree based on the set of records.

```
    for ta, data := range meta {
        ts[ta] = kdtree.New(data, false)
    }
```

Find collisions

Essentially a repeat of the read loop above, but instead of discordant reads select concordant reads.

```
const (
    filterReq  = boom.ProperPair
    filterMask = boom.Unmapped | boom.MateUnmapped | boom.Secondary | boom.Duplicate | filterReq
)
```

Provide a test for overlap at a specified genomic offset.

```
func overlap(a, b *illuminaRecord, at int) bool {
    return (a.A.Segment == b.A.Segment && a.A.End-at > b.A.Start && a.A.Start-at < b.A.End) ||
        (a.B.Segment == b.B.Segment && a.B.End-at > b.B.Start && a.B.Start-at < b.B.End) ||
        (a.A.Segment == b.B.Segment && a.A.End-at > b.B.Start && a.A.Start-at < b.B.End) ||
        (a.B.Segment == b.A.Segment && a.B.End-at > b.A.Start && a.B.Start-at < b.A.End)
}
```

Find collisions

```
t, ok := ts[tileAddress{ // Get the relevant tree.
    FlowCell: q.FlowCell,
    Lane:     q.Lane,
    Tile:     q.Tile,
}]
if !ok { // We didn't have one, so there is no closest polony.
    continue
}
n, d := t.Nearest(q)
if n == nil { // If there was a tree it must have a polony in it.
    panic("internal inconsistency: failed to find nearest")
}
nm := n.(*illuminaRecord)

if nm.Metadata == q.Metadata { // We only stored discordant, only queried concordant.
    panic("internal inconsistency: discordant pair is concordant pair‽")
}
if overlap(q, nm, offset) {
    coincident++
    fmt.Fprintf(os.Stderr, "@%d %0.fnm %+v -- %+v\n", offset, math.Sqrt(d), q, nm)
}
```

Output results

Output format for later analysis:

```
fmt.Printf("# %s\t%s\t%d\t%d\t%f\n",
    os.Args[1], readType, total, discordant, float64(discordant)/float64(total),
)
for i, off := range offsets {
    fmt.Printf("%s\t%s\t%s\t%d\t%f\n",
        os.Args[1], readType, off.label,
        coincident[i], float64(coincident[i])/float64(discordant),
    )
}
```

Harder problem

Can we compare the results for discordant reads to the situation with all mapped reads?

```
Function Collision(R reads):
    Let S be a spatially indexed store of reads
    Let C and M be 0
    For each read pair r in R:
        Increment M
        Store r in S
    For each read pair r in R:
        Let c be closest read to r in S
        If mapping-of(c) overlaps mapping-of(r)
            Increment C
    Return C, M.
```

This voids our original simplifying assumption:

- Much more data to store (discordant reads $\lesssim$ 5% of all reads)
- We now can't depend on the query not being in the store when we search

Memory load

Reduce weight of redundantly coded string data.

```
type Metadata struct {
    Type        Type
    Instrument  string     // Unique instrument name.
    Run         int        // Run id, -1 if not valid.
    FlowCell    string     // Flowcell id.
    Lane        int8       // Flowcell lane.
    Tile        int        // Tile number within the flowcell lane.
    Coordinate  Coordinate // Coordinate of the cluster within the tile.
    Mate        int8       // Member of a pair, 1 or 2 for paired reads.
    BadRead     bool       // Read failed filter.
    ControlBits int        // 0 when none of the control bits are on, otherwise it is an even num
    Multiplex   Multiplex  // Multiplexing information.
}

type Multiplex struct {
    Index int8   // Index is -1 if not valid.
    Tag   string // Tag is empty if not valid.
}
```

Intern strings

String values form a small set of unique values, but the Go runtime doesn't help here.

Define a helper type to store all the strings we've seen.

```
type store map[string]string
```

Convert a string to the representation we first saw. The garbage collector will clean up the redundant copies.

```
func (is store) intern(s string) string {
    if s == "" {
        return ""
    }
    t, ok := is[s]
    if ok {
        return t
    }
    is[s] = s
    return s
}
```

A query may match itself

The kdtree package helps here.

```
nk := kdtree.NewNKeeper(2)
```

Now we store the two closest polonies and only consider the second.

```
t.NearestSet(nk, q)
if nk.Heap[0].Comparable == nil {
    panic("internal inconsistency: failed to find nearest")
}
if nk.Heap[1].Comparable == nil {
    // The second ComparableDist is the infinite distance marker,
    // so there was only one spot on the tile! We are it.
    continue
}
nm := nk.Heap[1].Comparable.(*illuminaRecord)
d := nk.Heap[1].Dist

// Reset the keeper for the next query.
nk.Heap = nk.Heap[:1]
nk.Heap[0].Comparable = nil
nk.Heap[0].Dist = inf
```

Let's use these

Since we've voided our simplifying assumptions used in the original implementation, we may as well just perform the entire analysis using these extensions.

```
Function Collision(R reads):
    Let S be a spatially indexed store of reads
    Let T be a rich collection of polony collision statistics
    For each read pair r in R:
        Increment M
        Store r in S
    For each read pair r in R:
        Let c be closest read to r in S
        If mapping-of(c) overlaps mapping-of(r)
            T += get-stats(c, r) // magic function
    Return T, M.
```

And make use of Go's concurrency where possible.

Summary

Operation

Can perform analysis on BAM files with >100 million read pairs in reasonable time.

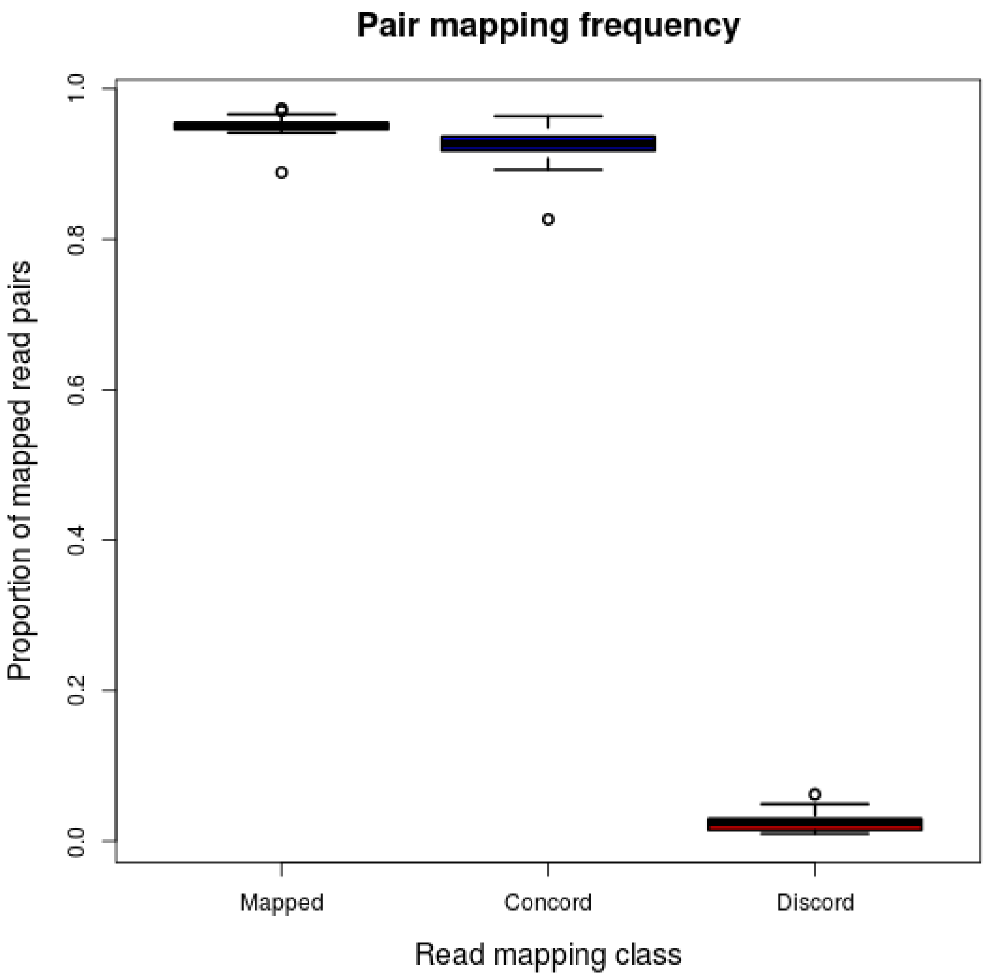
Discordant pair analysis requires ~3-4GB of system memory for 100 million pairs.

All-mapped pair analysis requires ~50GB for 100 million pairs.

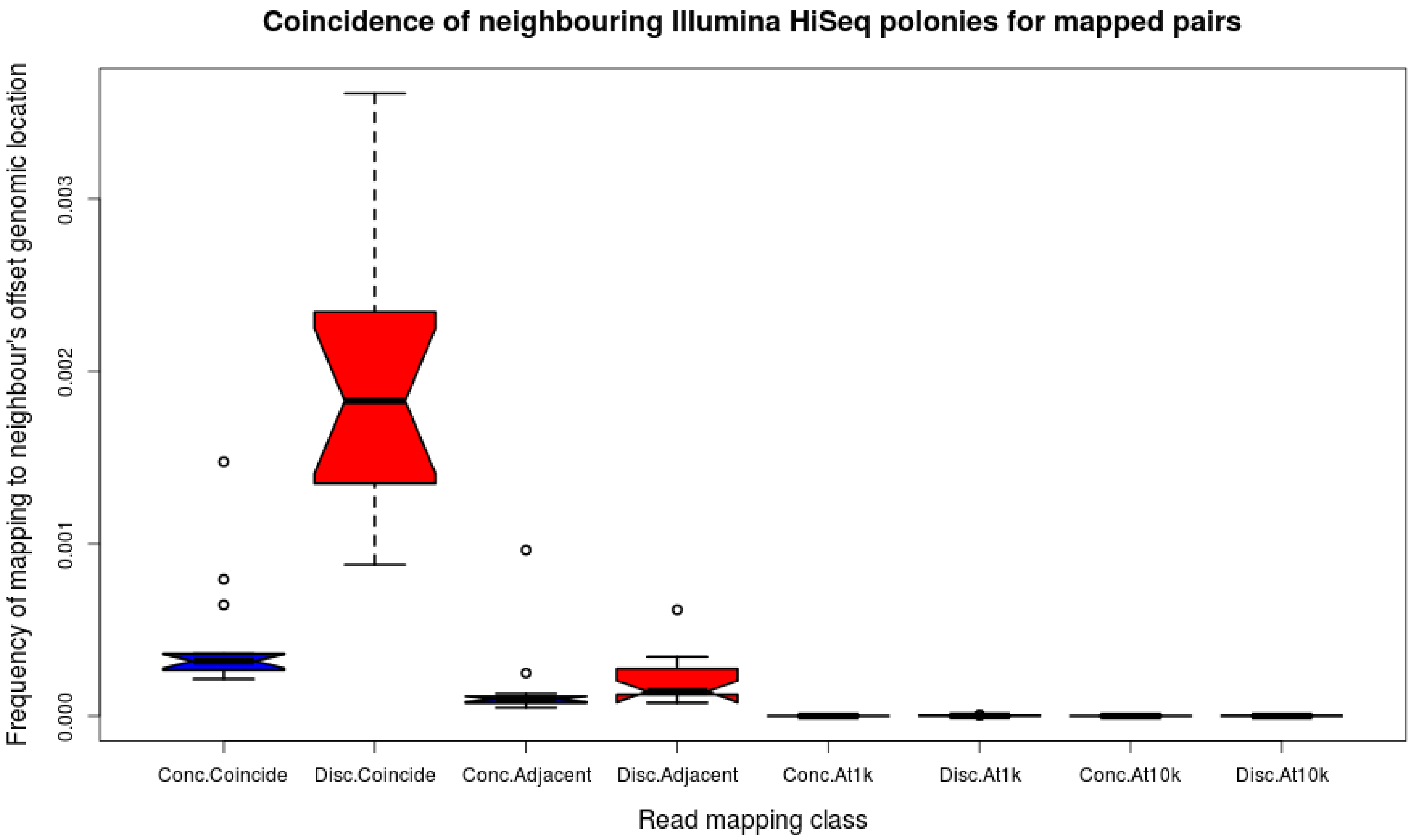
Two potential refinements for tree construction:

- A more memory-parsimonious approach can be easily implemented if the input is sorted by name to allow sequential analysis of independent tiles.
- Trees can be built concurrently, allowing parallel processing.

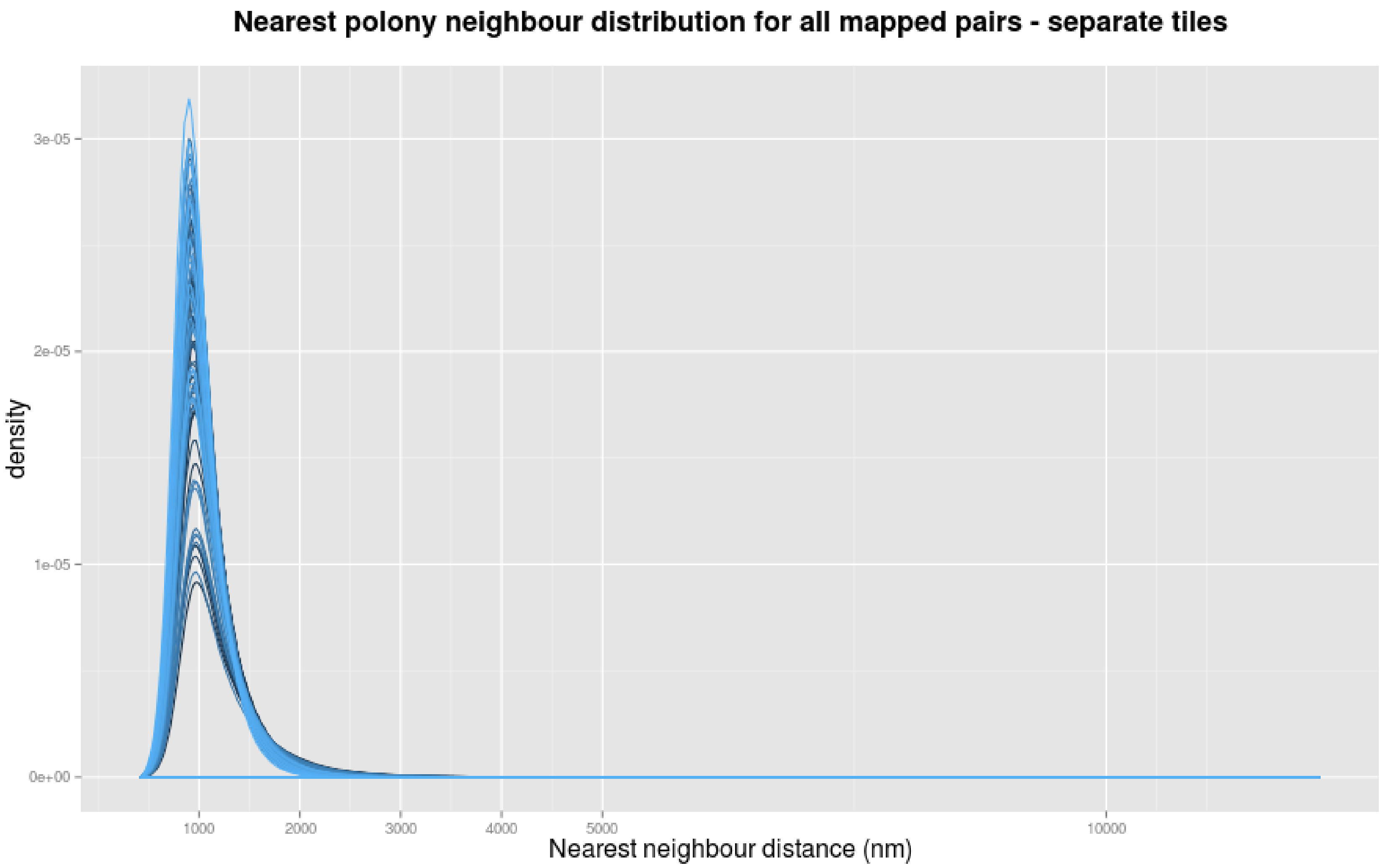
Results — mapping frequencies



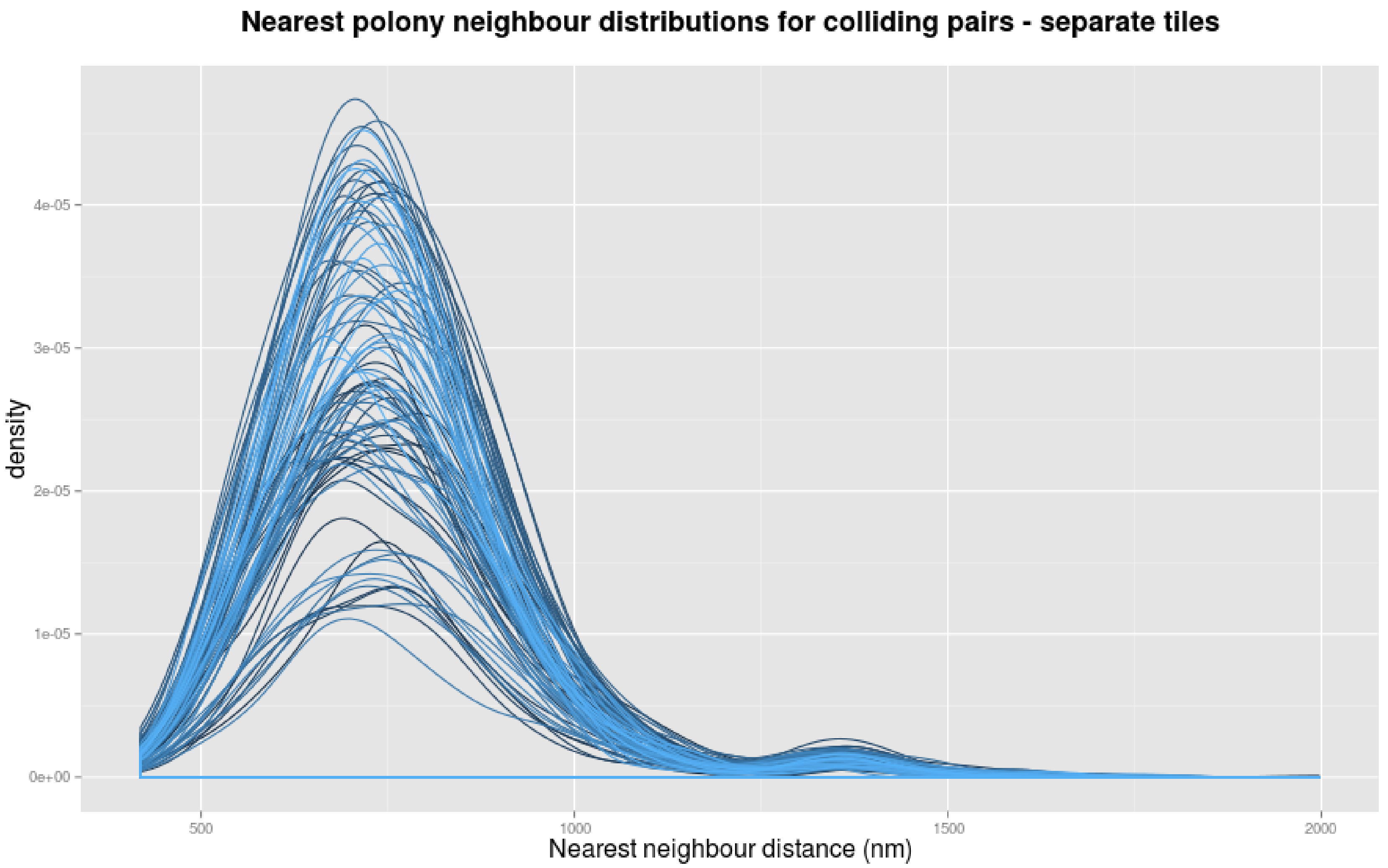
Results — collisions



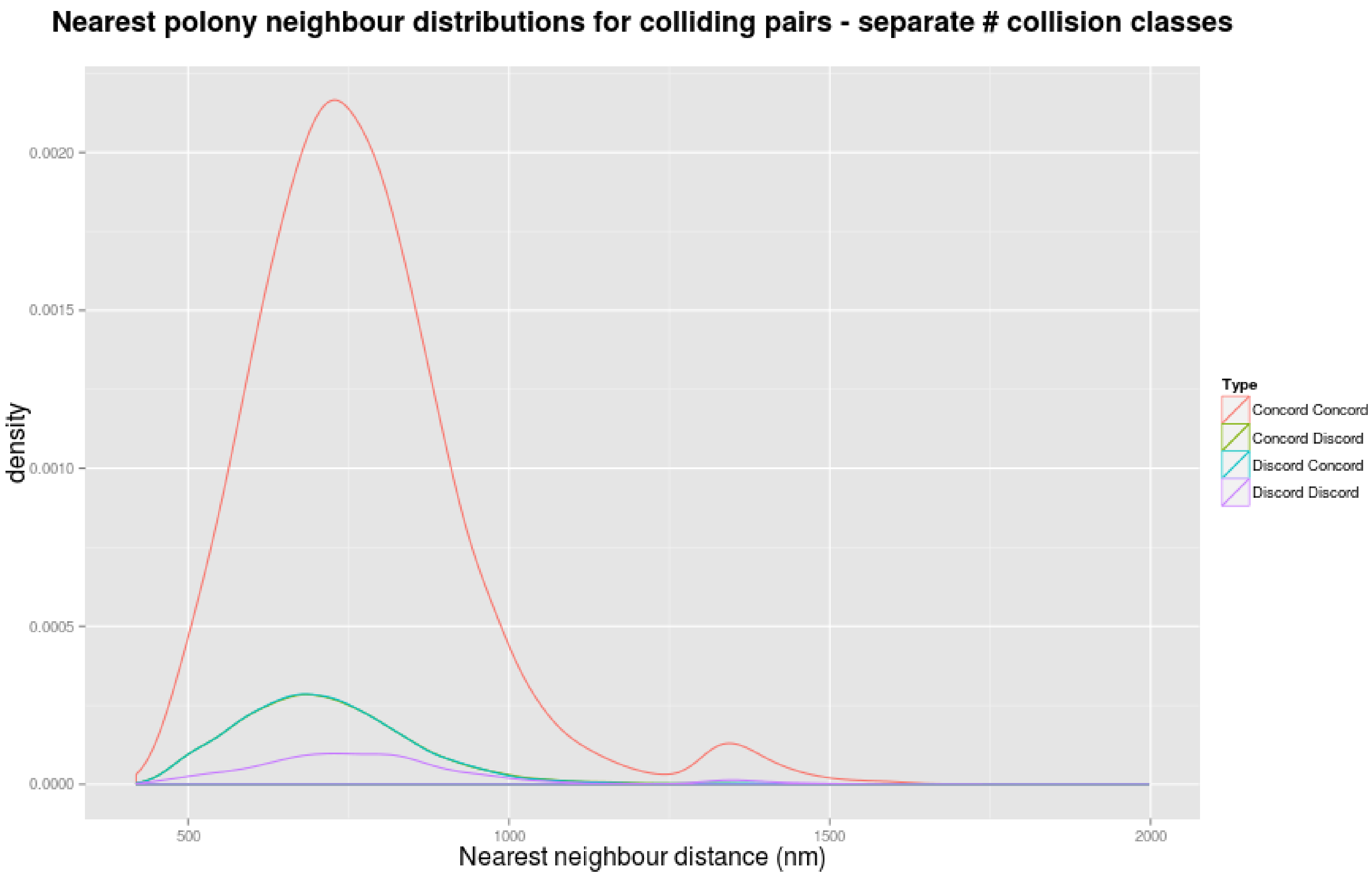
Results — density distributions - single experiment (91 tiles)



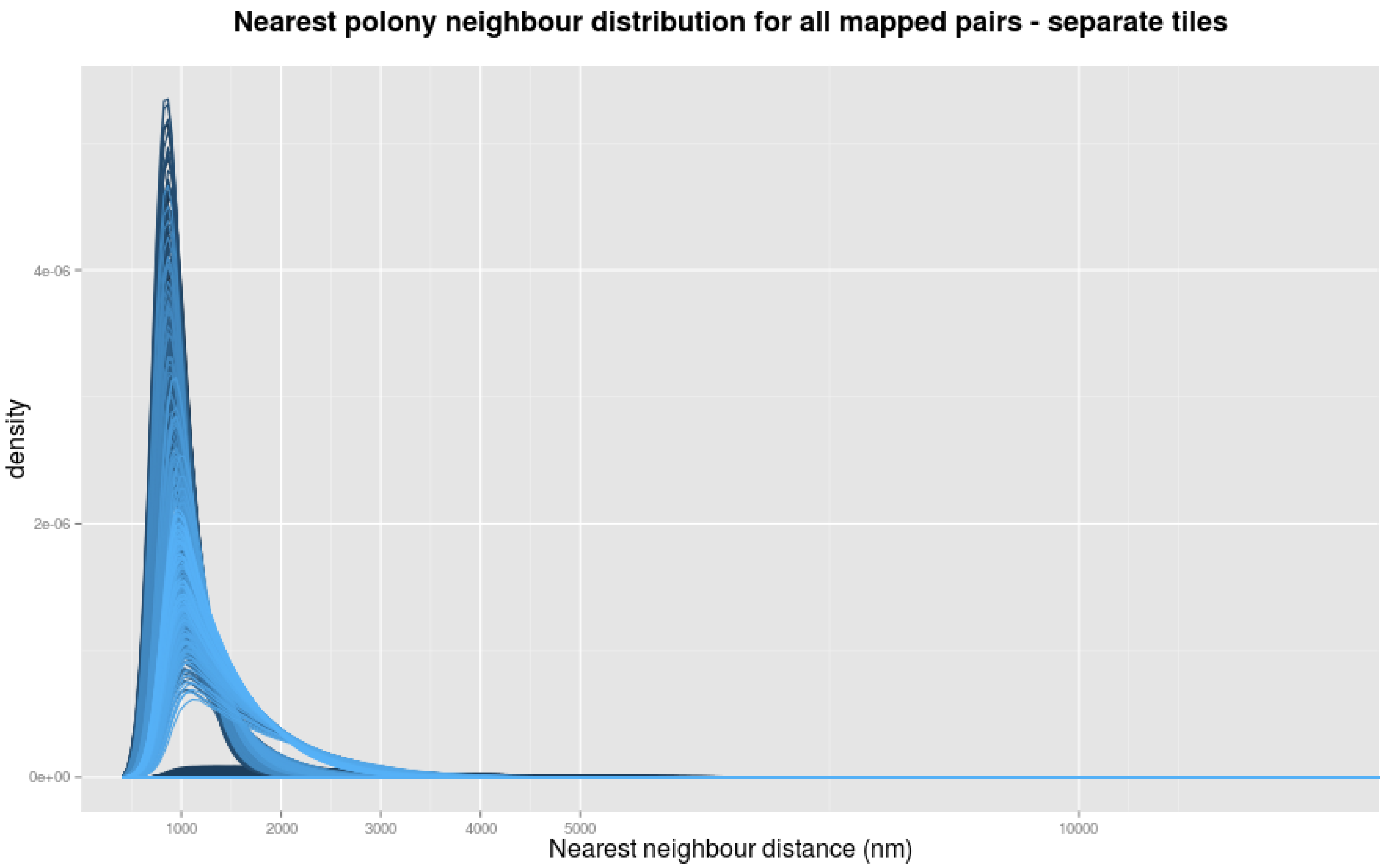
Results — density distributions - single experiment (91 tiles)



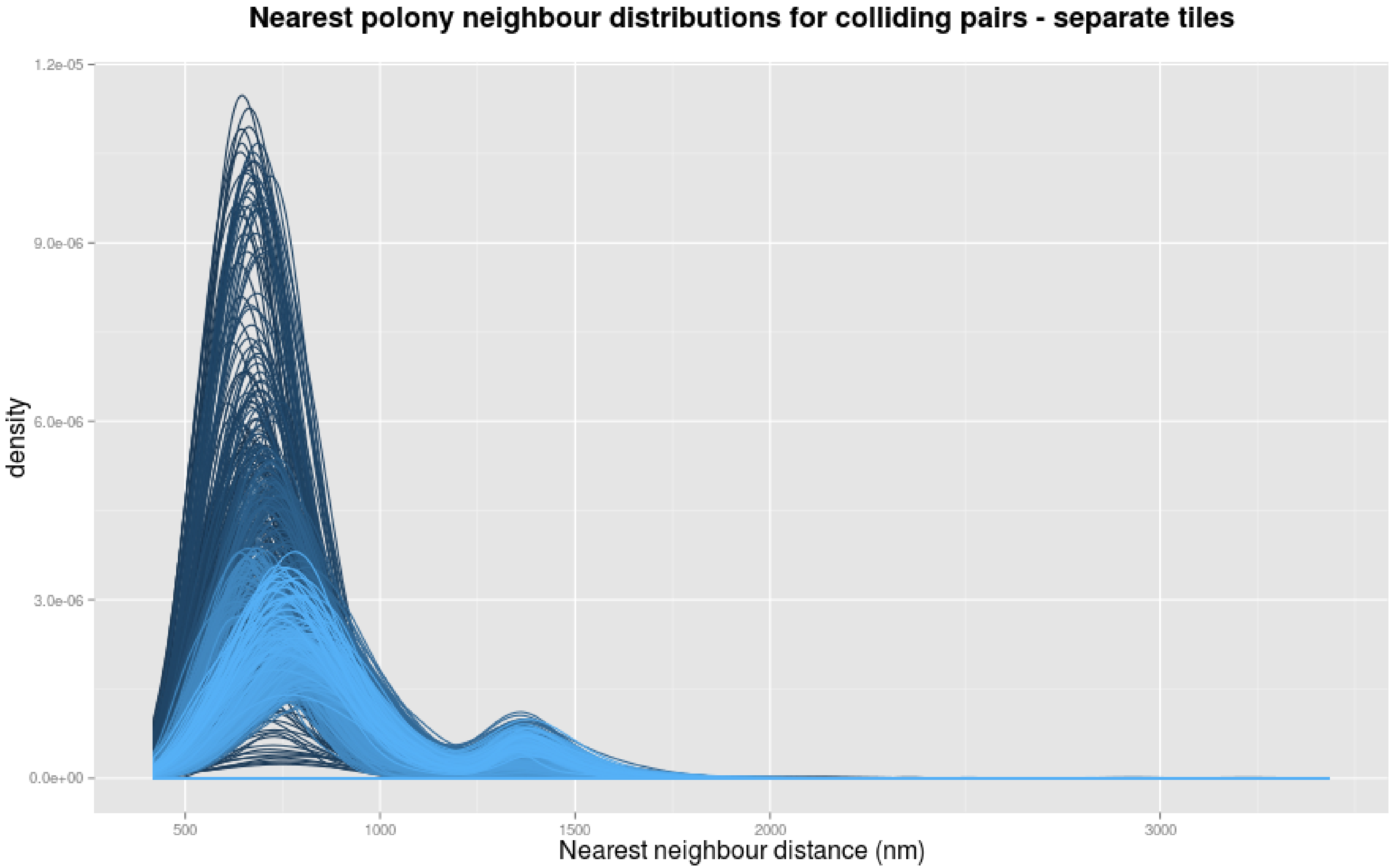
Results — density distributions - single experiment (91 tiles)



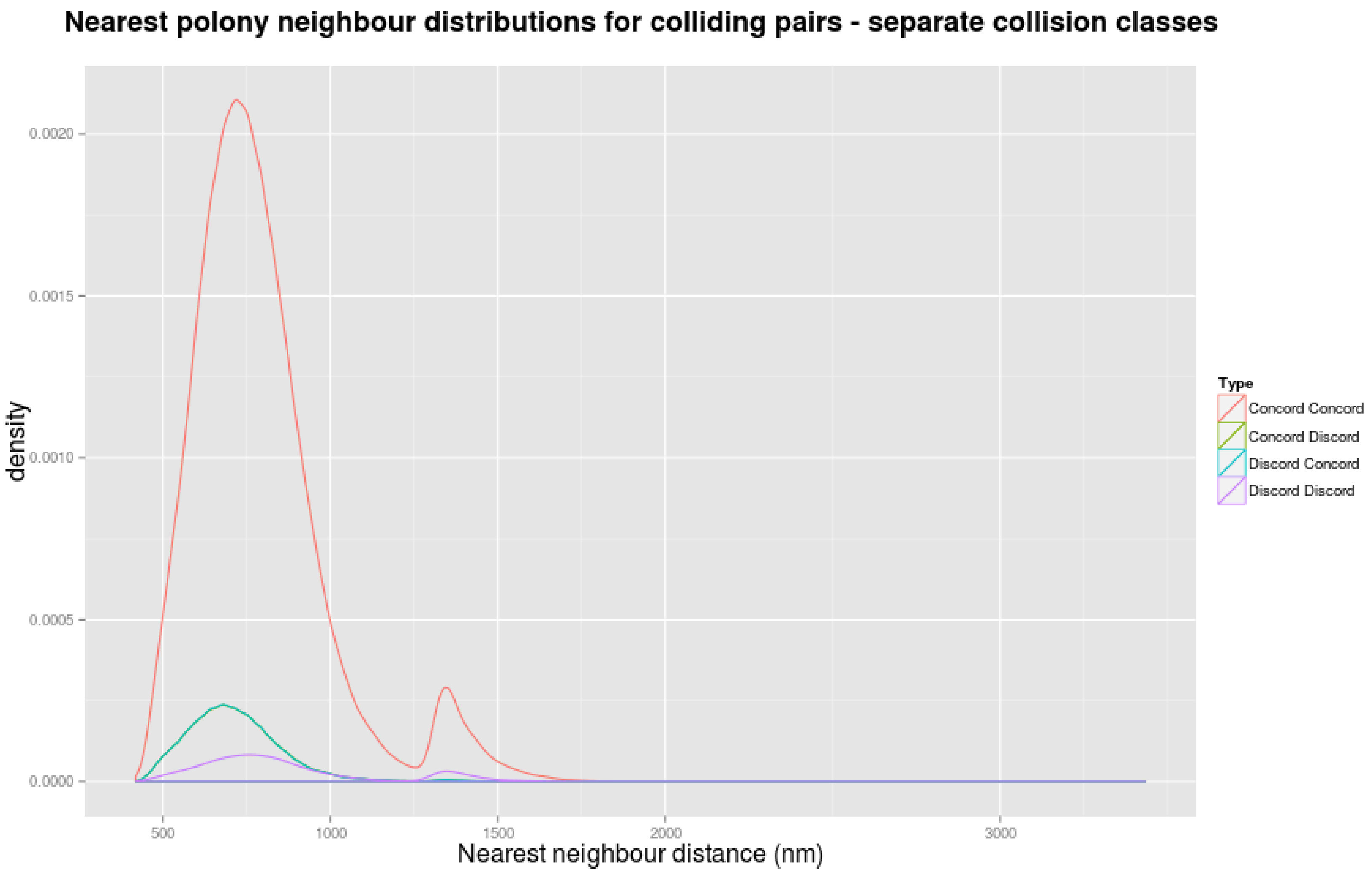
Results — density distributions - multiple experiments (876 tiles)



Results — density distributions - multiple experiments (876 tiles)



Results — density distributions - multiple experiments (876 tiles)



Interpretation

The elevation of collisions in the concordant class appears mainly due to nearly identical pairs.

- These are probably optical duplicates that are missed by our deduplication protocol which is based on perfect end matching (a translation of the picard algorithm).

The mechanism for generation of discordant pairs by collision is not yet clear.

These effects are low frequency, so largely not important — significant for people looking at low frequency structural variation events though.

Opportunities

Use to filter optical duplicates.

Thank you

Dan Kortschak
Bioinformatics Group
School of Molecular and Biomedical Science
The University of Adelaide
dan.kortschak@adelaide.edu.au (mailto:dan.kortschak@adelaide.edu.au)
http://code.google.com/p/biogo/ (http://code.google.com/p/biogo/)