Package 'StructuralVariantAnnotation'

November 14, 2025

Type Package

Title Variant annotations for structural variants

Version 1.27.0

Date 2024-04-23

Description StructuralVariantAnnotation provides a framework for analysis of structural variants within the Bioconductor ecosystem. This package contains contains useful helper functions for dealing with structural variants in VCF format. The packages contains functions for parsing VCFs from a number of popular callers as well as functions for dealing with breakpoints involving two separate genomic loci encoded as GRanges objects.

License GPL-3 + file LICENSE

Depends GenomicRanges, rtracklayer, VariantAnnotation, BiocGenerics, R (>= 4.1.0)

Imports assertthat, Biostrings, pwalign, stringr, dplyr, methods, rlang, GenomicFeatures, IRanges, S4Vectors, SummarizedExperiment, GenomeInfoDb,

Suggests ggplot2, devtools, testthat (>= 2.1.0), roxygen2, rmarkdown, tidyverse, knitr, ggbio, biovizBase, TxDb.Hsapiens.UCSC.hg19.knownGene, BSgenome.Hsapiens.UCSC.hg19,

RoxygenNote 7.1.1

Encoding UTF-8

VignetteBuilder knitr

biocViews DataImport, Sequencing, Annotation, Genetics, VariantAnnotation

git_url https://git.bioconductor.org/packages/StructuralVariantAnnotation

git_branch devel

git_last_commit 72cd374

git_last_commit_date 2025-10-29

Repository Bioconductor 3.23

Date/Publication 2025-11-13

Author Daniel Cameron [aut, cre] (ORCID:
<https://orcid.org/0000-0002-0951-7116>),
Ruining Dong [aut] (ORCID: <https://orcid.org/0000-0003-1433-0484>)

Maintainer Daniel Cameron <daniel.l.cameron@gmail.com>

Contents

`align_breakpoints` *Adjusting the nominal position of a pair of partnered breakpoint.*

Description

Adjusting the nominal position of a pair of partnered breakpoint.

Usage

```
align_breakpoints(
  vcf,
  align = c("centre"),
  is_higher_breakend = names(vcf) < info(vcf)$PARID
)
```

Arguments

vcf	A VCF object.
align	The alignment type.
is_higher_breakend	Breakpoint ID ordering.

Value

A VCF object with adjusted nominal positions.

---

breakendRanges *Extracting unpartnered breakend structural variants as a GRanges*

Description

Extracting unpartnered breakend structural variants as a GRanges

Usage

```
breakendRanges(x, ...)

## S4 method for signature 'VCF'
breakendRanges(x, ...)
```

Arguments

x	A VCF object.
...	Parameters of `.breakpointRanges()`. See breakpointRanges for more details.

Details

The VCF standard supports single breakends where a breakend is not part of a novel adjacency and lacks a mate. This function supports parsing single breakends to GRanges, where a dot symbol is used in the ALT field to annotate the directional information. Single breakends provide insights to situations when one side of the structural variant is not observed, due to e.g. low mappability, non-reference contigs, complex multi-break operations, etc. See Section 5.4.9 of https://samtools.github.io/hts-specs/VCFv4.3.pdf for details of single breakends.

Value

A GRanges object of SVs.

Methods (by class)

- VCF: Extracting unpartnered structural variants as GRanges.

Examples

```
vcf.file <- system.file("extdata", "gridss.vcf",
                        package = "StructuralVariantAnnotation")
vcf <- VariantAnnotation::readVcf(vcf.file, "hg19")
breakendRanges(vcf)
breakendRanges(vcf, nominalPosition=TRUE)
```

---

breakpointgr2bedpe *Converting breakpoint GRanges to BEDPE-like dataframe*

Description

Converting breakpoint GRanges to BEDPE-like dataframe

Usage

```
breakpointgr2bedpe(gr)
```

Arguments

gr	A GRanges object.

Details

breakpointgr2bedpe converts a breakpoint GRanges to a BEDPE-formatted dataframe. The BEDPE format consists of two sets of genomic loci, optional columns of name, score, strand1, strand2 and any user-defined fields. See https://bedtools.readthedocs.io/en/latest/content/general-usage.html for more details on the BEDPE format.

Value

A BEDPE-formatted data frame.

Examples

```
#coverting a GRanges object to BEDPE-like dataframe
vcf.file <- system.file("extdata", "gridss.vcf", package = "StructuralVariantAnnotation")
vcf <- VariantAnnotation::readVcf(vcf.file, "hg19")
gr <- breakpointRanges(vcf)
breakpointgr2bedpe(gr)
```

`breakpointgr2pairs` *Converts a breakpoint GRanges object to a Pairs object*

Description

Converts a breakpoint GRanges object to a Pairs object

Converts a BEDPE Pairs containing pairs of GRanges loaded using to a breakpoint GRanges object.

Usage

```
breakpointgr2pairs(
  bpgr,
  writeQualAsScore = TRUE,
  writeName = TRUE,
  bedpeName = NULL,
  firstInPair = NULL
)

pairs2breakpointgr(
  pairs,
  placeholderName = "bedpe",
  firstSuffix = "_1",
  secondSuffix = "_2",
  nameField = "name",
  renameScoreToQUAL = TRUE
)
```

Arguments

`bpgr`	breakpoint GRanges object
`writeQualAsScore`	write the breakpoint GRanges QUAL field as the score fields for compatibility with BEDPE rtracklayer export
`writeName`	write the breakpoint GRanges QUAL field as the score fields for compatibility with BEDPE rtracklayer export
`bedpeName`	function that returns the name to use for the breakpoint. Defaults to the sourceId, name column, or row names (in that priority) of the first breakend of each pair.
`firstInPair`	function that returns TRUE for breakends that are considered the first in the pair, and FALSE for the second in pair breakend. By default, the first in the pair is the breakend with the lower ordinal in the breakpoint GRanges object.
`pairs`	a Pairs object consisting of two parallel genomic loci.
`placeholderName`	prefix to use to ensure each entry has a unique ID.
`firstSuffix`	first in pair name suffix to ensure breakend name uniqueness

secondSuffix	second in pair name suffix to ensure breakend name uniqueness
nameField	Fallback field for row names if the Pairs object does not contain any names. BEDPE files loaded using rtracklayer use the "name" field.
renameScoreToQUAL	renames the 'score' column to 'QUAL'. Performing this rename results in a consistent variant quality score column name for variant loaded from BEDPE and VCF.

Details

Breakpoint-level column names will override breakend-level column names.

Value

Pairs GRanges object suitable for export to BEDPE by rtracklayer

Breakpoint GRanges object.

Examples

```
vcf.file <- system.file("extdata", "gridss.vcf", package = "StructuralVariantAnnotation")
bpgr <- breakpointRanges(VariantAnnotation::readVcf(vcf.file))
pairgr <- breakpointgr2pairs(bpgr)
#rtracklayer::export(pairgr, con="example.bedpe")
bedpe.file <- system.file("extdata", "gridss.bedpe", package = "StructuralVariantAnnotation")
bedpe.pairs <- rtracklayer::import(bedpe.file)
bedpe.bpgr <- pairs2breakpointgr(bedpe.pairs)
```

---

breakpointGRangesToVCF

Converts the given breakpoint GRanges object to VCF format in breakend notation.

Description

Converts the given breakpoint GRanges object to VCF format in breakend notation.

Usage

```
breakpointGRangesToVCF(gr, ...)
```

Arguments

gr	breakpoint GRanges object. Can contain both breakpoint and single breakend SV records.
...	For cbind and rbind a list of VCF objects. For all other methods ... are additional arguments passed to methods. See VCF class in VariantAnnotation for more details.

Value

A VCF object.

`breakpointRanges` *Extracting the structural variants as a GRanges.*

Description

Extracting the structural variants as a GRanges.

.breakpointRanges() is an internal function for extracting structural variants as GRanges.

Usage

```
breakpointRanges(x, ...)

## S4 method for signature 'VCF'
breakpointRanges(x, ...)

.breakpointRanges(
  vcf,
  nominalPosition = FALSE,
  placeholderName = "svrecord",
  suffix = "_bp",
  info_columns = NULL,
  unpartneredBreakends = FALSE,
  inferMissingBreakends = FALSE,
  ignoreUnknownSymbolicAlleles = FALSE
)
```

Arguments

`x`	A VCF object
`...`	Parameters of `.breakpointRanges()`. See below.
`vcf`	A VCF object.
`nominalPosition`	Determines whether to call the variant at the nominal VCF position, or to call the confidence interval (incorporating any homology present). Default value is set to FALSE, where the interval is called based on the CIPOS tag. When set to TRUE, the ranges field contains the nominal variant position only.
`placeholderName`	Variant name prefix to assign to unnamed variants.
`suffix`	The suffix to append to variant names.
`info_columns`	VCF INFO columns to include in the GRanges object.
`unpartneredBreakends`	Determining whether to report unpartnered breakends. Default is set to FALSE.

`inferMissingBreakends`
: Infer missing breakend records from ALT field of records without matching partners

`ignoreUnknownSymbolicAlleles`
: Ignore unknown symbolic alleles. StructuralVariantAnnotation currently handles INS, INV, DEL, DUP as well as the VCF specifications non-compliant RPL, TRA symbolic alleles.

Details

Structural variants are converted to breakend notation. Due to ambiguities in the VCF specifications, structural variants with multiple alt alleles are not supported. The CIPOS tag describes the uncertainty interval around the position of the breakend. See Section 5.4.8 of https://samtools.github.io/hts-specs/VCFv4.3.pdf for details of CIPOS. If HOMLEN or HOMSEQ is defined without CIPOS, it is assumed that the variant position is left aligned. A breakend on the '+' strand indicates a break immediately after the given position, to the left of which is the DNA segment involved in the breakpoint. The '-' strand indicates a break immediately before the given position, rightwards of which is the DNA segment involved in the breakpoint. Unpaired variants are removed at this stage.

Value

A GRanges object of SVs.

Methods (by class)

- `VCF`: Extracting structural variants as GRanges.

Examples

```
vcf.file <- system.file("extdata", "vcf4.2.example.sv.vcf",
                        package = "StructuralVariantAnnotation")
vcf <- VariantAnnotation::readVcf(vcf.file, "hg19")
breakpointRanges(vcf)
breakpointRanges(vcf, nominalPosition=TRUE)
```

`calculateReferenceHomology` *Calculates the length of inexact homology between the breakpoint sequence and the reference*

Description

Calculates the length of inexact homology between the breakpoint sequence and the reference

Usage

```
calculateReferenceHomology(
  gr,
  ref,
  anchorLength = 300,
  margin = 5,
  match = 2,
  mismatch = -6,
  gapOpening = 5,
  gapExtension = 3
)
```

Arguments

gr	reakpoint GRanges
ref	reference BSgenome
anchorLength	Number of bases to consider for homology
margin	Number of additional reference bases include. This allows for inexact homology to be detected even in the presence of indels.
match	see pwalign::pairwiseAlignment
mismatch	see pwalign::pairwiseAlignment
gapOpening	see pwalign::pairwiseAlignment
gapExtension	see pwalign::pairwiseAlignment

Value

A dataframe containing the length of inexact homology between the breakpoint sequence and the reference.

---

countBreakpointOverlaps

Counting overlapping breakpoints between two breakpoint sets

Description

Counting overlapping breakpoints between two breakpoint sets

Usage

```
countBreakpointOverlaps(
  querygr,
  subjectgr,
  countOnlyBest = FALSE,
  breakpointScoreColumn = "QUAL",
  maxgap = -1L,
```

```
  minoverlap = 0L,
  ignore.strand = FALSE,
  sizemargin = NULL,
  restrictMarginToSizeMultiple = NULL
)
```

Arguments

querygr, subjectgr, maxgap, minoverlap, ignore.strand, sizemargin, restrictMarginToSizeMultiple
: See findBreakpointOverlaps().

countOnlyBest
: Default value set to FALSE. When set to TRUE, the result count each subject breakpoint as overlaping only the best overlapping query breakpoint. The best breakpoint is considered to be the one with the highest QUAL score.

breakpointScoreColumn
: Query column defining a score for determining which query breakpoint is considered the best when countOnlyBest=TRUE.

Details

countBreakpointOverlaps() returns the number of overlaps between breakpoint objects, based on the output of findBreakpointOverlaps(). See GenomicRanges::countOverlaps-methods

Value

An integer vector containing the tabulated query overlap hits.

Examples

```
truth_vcf = VariantAnnotation::readVcf(system.file("extdata", "na12878_chr22_Sudmunt2015.vcf",
package = "StructuralVariantAnnotation"))
crest_vcf = VariantAnnotation::readVcf(system.file("extdata", "na12878_chr22_crest.vcf",
package = "StructuralVariantAnnotation"))
caller_bpgr = breakpointRanges(crest_vcf)
caller_bpgr$true_positive = countBreakpointOverlaps(caller_bpgr, breakpointRanges(truth_vcf),
  maxgap=100, sizemargin=0.25, restrictMarginToSizeMultiple=0.5, countOnlyBest=TRUE)
```

---

elementExtract *Extracts the element of each element at the given position*

Description

Extracts the element of each element at the given position

Usage

```
elementExtract(x, offset = 1)
```

Arguments

x	list-like object
offset	offset of list

Value

The element of each element at given positions.

`extractBreakpointSequence` *Extracts the breakpoint sequence.*

Description

Extracts the breakpoint sequence.

Usage

```
extractBreakpointSequence(gr, ref, anchoredBases, remoteBases = anchoredBases)
```

Arguments

gr	breakpoint GRanges
ref	Reference BSgenome
anchoredBases	Number of bases leading into breakpoint to extract
remoteBases	Number of bases from other side of breakpoint to extract

Details

The sequence is the sequenced traversed from the reference anchor bases to the breakpoint. For backward (-) breakpoints, this corresponds to the reverse compliment of the reference sequence bases.

Value

Breakpoint sequence around the variant position.

`extractReferenceSequence` *Returns the reference sequence around the breakpoint position*

Description

Returns the reference sequence around the breakpoint position

Usage

```
extractReferenceSequence(
  gr,
  ref,
  anchoredBases,
  followingBases = anchoredBases
)
```

Arguments

`gr`	breakpoint GRanges
`ref`	Reference BSgenome
`anchoredBases`	Number of bases leading into breakpoint to extract
`followingBases`	Number of reference bases past breakpoint to extract

Details

The sequence is the sequenced traversed from the reference anchor bases to the breakpoint. For backward (-) breakpoints, this corresponds to the reverse compliment of the reference sequence bases.

Value

Reference sequence around the breakpoint position.

`findBreakpointOverlaps` *Finding overlapping breakpoints between two breakpoint sets*

Description

Finding overlapping breakpoints between two breakpoint sets

Usage

```
findBreakpointOverlaps(
  query,
  subject,
  maxgap = -1L,
  minoverlap = 0L,
  ignore.strand = FALSE,
  sizemargin = NULL,
  restrictMarginToSizeMultiple = NULL
)
```

Arguments

query, subject	Both of the input objects should be GRanges objects. Unlike findOverlaps(), subject cannot be ommitted. Each breakpoint must be accompanied with a partner breakend, which is also in the GRanges, with the partner's id recorded in the partner field. See GenomicRanges::findOverlaps-methods for details.
maxgap, minoverlap	Valid overlapping thresholds of a maximum gap and a minimum overlapping positions between breakend intervals. Both should be scalar integers. maxgap allows non-negative values, and minoverlap allows positive values. See GenomicRanges::findOverlaps-methods for details.
ignore.strand	Default value is FALSE. strand information is ignored when set to TRUE. See GenomicRanges::findOverlaps-methods for details.
sizemargin	Error margin in allowable size to prevent matching of events of different sizes, e.g. a 200bp event matching a 1bp event when maxgap is set to 200.
restrictMarginToSizeMultiple	Size restriction multiplier on event size. The default value of 0.5 requires that the breakpoint positions can be off by at maximum, half the event size. This ensures that small deletion do actually overlap at least one base pair.

Details

findBreakpointOverlaps() is an efficient adaptation of findOverlaps-methods() for breakend ranges. It searches for overlaps between breakpoint objects, and return a matrix including index of overlapping ranges as well as error stats. All breakends must have their partner breakend included in the partner field. A valid overlap requires that breakends on boths sides meets the overlapping requirements.

See GenomicRanges::findOverlaps-methods for details of overlap calculation.

Value

A dataframe containing index and error stats of overlapping breakpoints.

Examples

```
#reading in VCF files
query.file <- system.file("extdata", "gridss-na12878.vcf", package = "StructuralVariantAnnotation")
```

```
subject.file <- system.file("extdata", "gridss.vcf", package = "StructuralVariantAnnotation")
query.vcf <- VariantAnnotation::readVcf(query.file, "hg19")
subject.vcf <- VariantAnnotation::readVcf(subject.file, "hg19")
#parsing vcfs to GRanges objects
query.gr <- breakpointRanges(query.vcf)
subject.gr <- breakpointRanges(subject.vcf)
#find overlapping breakpoint intervals
findBreakpointOverlaps(query.gr, subject.gr)
findBreakpointOverlaps(query.gr, subject.gr, ignore.strand=TRUE)
findBreakpointOverlaps(query.gr, subject.gr, maxgap=100, sizemargin=0.5)
```

`findInsDupOverlaps` *Finds duplication events that are reported as inserts. As sequence alignment algorithms do no allow backtracking, long read-based variant callers will frequently report small duplication as insertion events. Whilst both the duplication and insertion representations result in the same sequence, this representational difference is problematic when comparing variant call sets.*

Description

WARNING: this method does not check that the inserted sequence actually matched the duplicated sequence.

Usage

```
findInsDupOverlaps(query, subject, maxgap = -1L, maxsizedifference = 0L)
```

Arguments

`query`	a breakpoint GRanges object
`subject`	a breakpoint GRanges object
`maxgap`	maximum distance between the insertion position and the duplication
`maxsizedifference`	maximum size difference between the duplication and insertion.

Value

Hits object containing the ordinals of the matching breakends in the query and subject

`findTransitiveCalls` *Identifies potential transitive imprecise calls that can be explained by traversing multiple breakpoints.*

Description

Transitive calls are imprecise breakpoints or breakpoints with inserted sequence that can be explained by a sequence of breakpoints. That is, A-C calls in which additional sequence may be between A and C that can be explained by A-B-C.

Usage

```
findTransitiveCalls(
  transitiveGr,
  subjectGr,
  maximumImpreciseInsertSize = 700,
  minimumTraversedBreakpoints = 2,
  maximumTraversedBreakpoints = 6,
  positionalMargin = 8,
  insertionLengthMargin = 50,
  insLen = transitiveGr$insLen,
 impreciseTransitiveCalls = (transitiveGr$HOMLEN == 0 | is.null(transitiveGr$HOMLEN))
    & start(transitiveGr) != end(transitiveGr),
  impreciseSubjectCalls = (subjectGr$HOMLEN == 0 | is.null(subjectGr$HOMLEN)) &
    start(subjectGr) != end(subjectGr),
  allowImprecise = FALSE
)
```

Arguments

`transitiveGr`	a breakpoint GRanges object containing imprecise calls
`subjectGr`	breakpoints to traverse
`maximumImpreciseInsertSize`	Expected number of bases to traverse imprecise calls.
`minimumTraversedBreakpoints`	Minimum number of traversed breakpoints to consider a transitive
`maximumTraversedBreakpoints`	Maximum number of breakpoints to traverse when looking for an explanation of the transitive calls
`positionalMargin`	Allowable margin of error when matching call positional overlaps. A non-zero margin allows for matching of breakpoint with imperfect homology.
`insertionLengthMargin`	Allowable difference in length between the inserted sequence and the traversed path length. Defaults to 50bp to allow for long read indel errors.

insLen Integer vector of same length as ‘transitiveGr‘ indicating the number of bases inserted at the breakpoint.

Defaults to transitiveGr$insLen which will be present if the GRanges was loaded from a VCF using breakpointRanges()

impreciseTransitiveCalls
Boolean vector of same length as ‘transitiveGr‘ indicating which calls are imprecise calls. Defaults to calls with a non-zero interval size that have no homology.

impreciseSubjectCalls
Boolean vector of same length as ‘subjectGr‘ indicating which calls are imprecise calls. Defaults to calls with a non-zero interval size that have no homology.

allowImprecise Allow traversal of imprecise calls. Defaults to FALSE as to prevent spurious results which skip some breakpoints when traversing multiple breakpoints E.g. An A-D transitive from an underlying A-B-C-D rearrangement will include A-B-D and A-C-D results if allowImprecise=TRUE.

Value

‘DataFrame‘ containing the transitive calls traversed with the following columns: | column | meaning | | —— | ——- | | transitive_breakpoint_name | Name of the transitive breakpoint a path was found for | | total_distance | Total length (in bp) of the path | | traversed_breakpoint_names | ‘CharacterList‘ of names of breakpoint traversed in the path | | distance_to_traversed_breakpoint | ‘IntegerList‘ of distances from start of path to end of traversing breakpoint |

---

hasPartner *Determines whether this breakend has a valid partner in this GRanges*

Description

Determines whether this breakend has a valid partner in this GRanges

Usage

```
hasPartner(gr, selfPartnerSingleBreakends = FALSE)
```

Arguments

gr GRanges object of SV breakends

selfPartnerSingleBreakends
treat single breakends as their own partner.

Value

True/False for each row in the breakpoint GRanges

Examples

```
#Subset to chromosome 6 intra-chromosomal events \code{vcf}
vcf.file <- system.file("extdata", "COLO829T.purple.sv.ann.vcf.gz",
  package = "StructuralVariantAnnotation")
vcf <- VariantAnnotation::readVcf(vcf.file)
gr <- breakpointRanges(vcf)
gr <- gr[seqnames(gr) == "6"]
# We now need to filter out inter-chromosomal events to ensure
# our GRanges doesn't contain any breakpoints whose partner
# has already been filtered out and no longer exists in the GRanges.
gr <- gr[hasPartner(gr)]
```

---

isStructural *Determining whether the variant is a structural variant*

Description

Determining whether the variant is a structural variant

Usage

```
isStructural(x, ...)

## S4 method for signature 'CollapsedVCF'
isStructural(x, ..., singleAltOnly = TRUE)

## S4 method for signature 'ExpandedVCF'
isStructural(x, ...)

## S4 method for signature 'VCF'
isStructural(x, ...)
```

Arguments

x	A VCF object.
...	Internal parameters.
singleAltOnly	Whether only single ALT values are accepted. Default is set to TRUE.

Details

The function takes a VCF object as input, and returns a logical value for each row, determining whether the variant is a structural variant.

Value

A logical list of which the length is the same with the input object.

Methods (by class)

- `CollapsedVCF`: Determining whether a CollapsedVCF object is a strucrual variant. Only single ALT values are accepted.
- `ExpandedVCF`: Determining whether a ExpandedVCF object is a structural variant.
- `VCF`: Determining whether a VCF object is a structural variant.

Examples

```
vcf.file <- system.file("extdata", "gridss.vcf", package = "StructuralVariantAnnotation")
vcf <- VariantAnnotation::readVcf(vcf.file, "hg19")
isStructural(vcf)
```

`isSymbolic` *Determining whether the variant is a symbolic allele.*

Description

Determining whether the variant is a symbolic allele.

Usage

```
isSymbolic(x, ...)

## S4 method for signature 'CollapsedVCF'
isSymbolic(x, ..., singleAltOnly = TRUE)

## S4 method for signature 'ExpandedVCF'
isSymbolic(x, ...)
```

Arguments

`x`	A VCF object.
`...`	Internal parameters.
`singleAltOnly`	Whether only single ALT values are accepted. Default is set to TRUE.

Details

The function takes a VCF object as input, and returns a logical value for each row, determining whether the variant is a symbolic allele.

Value

A logical list of which the length is the same with the input object.

Methods (by class)

- `CollapsedVCF`: Determining whether a CollapsedVCF object is a symbolic allele. Only single ALT values are accepted.
- `ExpandedVCF`: Determining whether a ExpandedVCF object is a symbolic allele

Examples

```
vcf.file <- system.file("extdata", "gridss.vcf", package = "StructuralVariantAnnotation")
vcf <- VariantAnnotation::readVcf(vcf.file, "hg19")
isSymbolic(vcf)
```

---

numtDetect *Detecting nuclear mitochondria fusion events.*

Description

Detecting nuclear mitochondria fusion events.

Usage

```
numtDetect(gr, nonStandardChromosomes = FALSE, max_ins_dist = 1000)
```

Arguments

`gr`	A GRanges object
`nonStandardChromosomes`	Whether to report insertion sites on non-standard reference chromosomes. Default value is set to FALSE.
`max_ins_dist`	The maxium distance allowed on the reference genome between the paired insertion sites. Only intra-chromosomal NUMT events are supported. Default value is 1000.

Details

Nuclear mitochondrial fusion (NUMT) is a common event found in human genomes. This function searches for NUMT events by identifying breakpoints supporting the fusion of nuclear chromosome and mitochondrial genome. Only BND notations are supported at the current stage. Possible linked nuclear insertion sites are reported using SV IDs in the candidatePartnerId metadata column.

Value

A GRanges object of possible NUMT loci.

Examples

```
vcf.file <- system.file("extdata", "MT.vcf", package = "StructuralVariantAnnotation")
vcf <- VariantAnnotation::readVcf(vcf.file, "hg19")
gr <- breakpointRanges(vcf, nominalPosition=TRUE)
numt.gr <- numtDetect(gr)
```

partner *GRanges representing the breakend coordinates of structural variants #@export Partner breakend for each breakend.*

Description

GRanges representing the breakend coordinates of structural variants #@export Partner breakend for each breakend.

Usage

```
partner(gr, selfPartnerSingleBreakends = FALSE)
```

Arguments

gr	GRanges object of SV breakends
selfPartnerSingleBreakends	treat single breakends as their own partner.

Details

All breakends must have their partner breakend included in the GRanges.

Value

A GRanges object in which each entry is the partner breakend of those in the input object.

Examples

```
#reading in a VCF file as \code{vcf}
vcf.file <- system.file("extdata", "gridss.vcf", package = "StructuralVariantAnnotation")
vcf <- VariantAnnotation::readVcf(vcf.file, "hg19")
#parsing \code{vcf} to GRanges object \code{gr}
gr <- breakpointRanges(vcf)
#output partner breakend of each breakend in \code{gr}
partner(gr)
```

rtDetect *Detecting retrotranscript insertion in nuclear genomes.*

Description

Detecting retrotranscript insertion in nuclear genomes.

Usage

```
rtDetect(gr, genes, maxgap = 100, minscore = 0.3)
```

Arguments

gr	A GRanges object
genes	TxDb object of genes. hg19 and hg38 are supported in the current version.
maxgap	The maxium distance allowed on the reference genome between the paired exon boundries.
minscore	The minimum proportion of intronic deletions of a transcript should be identified.

Details

This function searches for retroposed transcripts by identifying breakpoints supporting intronic deletions and fusions between exons and remote loci. Only BND notations are supported at the current stage.

Value

A GRangesList object, named insSite and rt, reporting breakpoints supporting insert sites and retroposed transcripts respectively. 'exon' and 'txs' in the metadata columns report exon_id and transcript_name from the 'genes' object.

---

simpleEventLength *Length of event if interpreted as an isolated breakpoint.*

Description

Length of event if interpreted as an isolated breakpoint.

Usage

```
simpleEventLength(gr)
```

Arguments

gr	breakpoint GRanges object

Value

Length of the simplest explanation of this breakpoint/breakend.

`simpleEventType` *Type of simplest explanation of event. Possible types are: | Type | Description | | BND | Single breakend | | CTX | Interchromosomal translocation | | INV | Inversion. | | DUP | Tandem duplication | | INS | Insertion | | DEL | Deletion |*

Description

Note that both ++ and – breakpoint will be classified as inversions regardless of whether both breakpoint that consistitute an actual inversion exists or not

Usage

```
simpleEventType(gr, insertionLengthThreshold = 0.5)
```

Arguments

`gr`	breakpoint GRanges object
`insertionLengthThreshold`	portion of inserted bases compared to total event size to be classified as an insertion. For example, a 5bp deletion with 5 inserted bases will be classified as an INS event.

Value

Type of simplest explanation of event

`StructuralVariantAnnotation` *StructuralVariantAnnotation: a package for SV annotation*

Description

StructuralVariantAnnotation contains useful helper functions for reading and interpreting structural variants calls. The packages contains functions for parsing VCFs from a number of popular caller as well as functions for dealing with breakpoints involving two separate genomic loci. The package takes a ‘GRanges‘ based breakend-centric approach.

Details

* Parse VCF objects with the ‘breakpointRanges()‘ and ‘breakendRanges()‘functions. * Find breakpoint overlaps with the ‘findBreakpointOverlaps()‘ and ‘countBreakpointOverlaps()‘ functions. * Generate BEDPE files for circos plot with ‘breakpointgr2pairs()‘ function. * ...

For more details on the features of StructuralVariantAnnotation, read the vignette: ‘browseVignettes(package = "StructuralVariantAnnotation")‘

Index