Package: ivs (via r-universe)

June 14, 2024

Title Interval Vectors

Version 0.2.0.9000

Description Provides a library for generic interval manipulations using a new interval vector class. Capabilities include: locating various kinds of relationships between two interval vectors, merging overlaps within a single interval vector, splitting an interval vector on its overlapping endpoints, and applying set theoretical operations on interval vectors. Many of the operations in this package were inspired by James Allen's interval algebra, Allen (1983) <doi:10.1145/182.358434>.

License MIT + file LICENSE

URL https://github.com/DavisVaughan/ivs, https://davisvaughan.github.io/ivs/

BugReports https://github.com/DavisVaughan/ivs/issues

Depends R (>= 3.5.0)

Imports glue (>= 1.6.2), lifecycle (>= 1.0.3), rlang (>= 1.1.0), vctrs (>= 0.6.0)

Suggests bit64 (>= 4.0.5), clock (>= 0.6.0), covr, dplyr (>= 1.1.0), knitr, rmarkdown, testthat (>= 3.0.0), tidyr (>= 1.1.4)

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Repository https://davisvaughan.r-universe.dev

RemoteUrl https://github.com/davisvaughan/ivs

RemoteRef HEAD

RemoteSha a93a4467a6f15e6831d16908ae97f9adbd209e0e

Contents

`allen-relation-count` *Count relations from Allen's Interval Algebra*

Description

`iv_count_relates()` is similar to `iv_count_overlaps()`, but it counts a specific set of relations developed by James Allen in the paper: Maintaining Knowledge about Temporal Intervals.

Usage

```
iv_count_relates(
  needles,
  haystack,
  ...,
  type,
  missing = "equals",
```

```
  no_match = 0L
)
```

Arguments

needles, haystack

[iv]

Interval vectors used for relation matching.

- Each element of needles represents the interval to search for.
- haystack represents the intervals to search in.

Prior to comparison, needles and haystack are coerced to the same type.

...

These dots are for future extensions and must be empty.

type

[character(1)]

The type of relationship to find. See the Allen's Interval Algebra section for a complete description of each type. One of:

- "precedes"
- "preceded-by"
- "meets"
- "met-by"
- "overlaps"
- "overlapped-by"
- "starts"
- "started-by"
- "during"
- "contains"
- "finishes"
- "finished-by"
- "equals"

missing

[integer(1) / "equals" / "error"]

Handling of missing intervals in needles.

- "equals" considers missing intervals in needles as exactly equal to missing intervals in haystack when determining if there is a matching relationship between them.
- "error" throws an error if any intervals in needles are missing.
- If a single integer value is provided, this represents the count returned for a missing interval in needles. Use 0L to force missing intervals to never match.

no_match

[integer(1) / "error"]

Handling of needles without a match.

- "error" throws an error if any needles have zero matches.
- If a single integer is provided, this represents the count returned for a needle with zero matches. The default value gives unmatched needles a count of 0L.

Value

An integer vector the same size as `needles`.

Allen's Interval Algebra

The interval algebra developed by James Allen serves as a basis and inspiration for `iv_locate_overlaps()`, `iv_locate_precedes()`, and `iv_locate_follows()`. The original algebra is composed of 13 relations which have the following properties:

- Distinct: No pair of intervals can be related by more than one `type`.
- Exhaustive: All pairs of intervals are described by one of the `types`.
- Qualitative: No numeric intervals are considered. The relationships are computed by purely qualitative means.

Take the notation that `x` and `y` represent two intervals. Now assume that `x` can be represented as `[x_s, x_e)`, where `x_s` is the start of the interval and `x_e` is the end of it. Additionally, assume that `x_s < x_e`. With this notation, the 13 relations are as follows:

- *Precedes*:
  `x_e < y_s`
- *Preceded-by*:
  `x_s > y_e`
- *Meets*:
  `x_e == y_s`
- *Met-by*:
  `x_s == y_e`
- *Overlaps*:
  `(x_s < y_s) & (x_e > y_s) & (x_e < y_e)`
- *Overlapped-by*:
  `(x_e > y_e) & (x_s < y_e) & (x_s > y_s)`
- *Starts*:
  `(x_s == y_s) & (x_e < y_e)`
- *Started-by*:
  `(x_s == y_s) & (x_e > y_e)`
- *Finishes*:
  `(x_s > y_s) & (x_e == y_e)`
- *Finished-by*:
  `(x_s < y_s) & (x_e == y_e)`
- *During*:
  `(x_s > y_s) & (x_e < y_e)`
- *Contains*:
  `(x_s < y_s) & (x_e > y_e)`

- *Equals*:
  `(x_s == y_s) & (x_e == y_e)`

Note that when `missing = "equals"`, missing intervals will only match the `type = "equals"` relation. This ensures that the distinct property of the algebra is maintained.

Connection to other functions:

Note that some of the above relations are fairly restrictive. For example, `"overlaps"` only detects cases where `x` straddles `y_s`. It does not consider the case where `x` and `y` are equal to be an overlap (as this is `"equals"`) nor does it consider when `x` straddles `y_e` to be an overlap (as this is `"overlapped-by"`). This makes the relations extremely useful from a theoretical perspective, because they can be combined without fear of duplicating relations, but they don't match our typical expectations for what an "overlap" is.

`iv_locate_overlaps()`, `iv_locate_precedes()`, and `iv_locate_follows()` use more intuitive types that aren't distinct, but typically match your expectations better. They can each be expressed in terms of Allen's relations:

- `iv_locate_overlaps()`:
  - `"any"`:
    `overlaps | overlapped-by | starts | started-by | finishes | finished-by | during | contains | equals`
  - `"contains"`:
    `contains | started-by | finished-by | equals`
  - `"within"`:
    `during | starts | finishes | equals`
  - `"starts"`:
    `starts | started-by | equals`
  - `"ends"`:
    `finishes | finished-by | equals`
  - `"equals"`:
    `equals`
- `iv_locate_precedes()`:
  `precedes | meets`
- `iv_locate_follows()`:
  `preceded-by | met-by`

See Also

Locating relations from Allen's Interval Algebra

Examples

```
x <- iv(1, 3)
y <- iv(3, 4)

# `"precedes"` is strict, and doesn't let the endpoints match
iv_count_relates(x, y, type = "precedes")

# Since that is what `"meets"` represents
```

```
iv_count_relates(x, y, type = "meets")

# `"overlaps"` is a very specific type of overlap where an interval in
# `needles` straddles the start of an interval in `haystack`
x <- iv_pairs(c(1, 4), c(1, 3), c(0, 3), c(2, 5))
y <- iv(1, 4)

# It doesn't match equality, or when the starts match, or when the end
# of the interval in `haystack` is straddled instead
iv_count_relates(x, y, type = "overlaps")
```

`allen-relation-detect` *Detect relations from Allen's Interval Algebra*

Description

`iv_relates()` is similar to `iv_overlaps()`, but it detects a specific set of relations developed by James Allen in the paper: Maintaining Knowledge about Temporal Intervals.

Usage

```
iv_relates(needles, haystack, ..., type, missing = "equals")
```

Arguments

`needles`, `haystack`

`[iv]`

Interval vectors used for relation matching.

- Each element of `needles` represents the interval to search for.
- `haystack` represents the intervals to search in.

Prior to comparison, `needles` and `haystack` are coerced to the same type.

`...` These dots are for future extensions and must be empty.

`type` `[character(1)]`

The type of relationship to find. See the Allen's Interval Algebra section for a complete description of each type. One of:

- `"precedes"`
- `"preceded-by"`
- `"meets"`
- `"met-by"`
- `"overlaps"`
- `"overlapped-by"`
- `"starts"`
- `"started-by"`
- `"during"`
- `"contains"`

- "finishes"
- "finished-by"
- "equals"

missing [logical(1) / "equals" / "error"]

Handling of missing intervals in needles.

- "equals" considers missing intervals in needles as exactly equal to missing intervals in haystack when determining if there is a matching relationship between them. Matched missing intervals in needles result in a TRUE value in the result, and unmatched missing intervals result in a FALSE value.
- "error" throws an error if any intervals in needles are missing.
- If a single logical value is provided, this represents the value returned in the result for intervals in needles that are missing. You can force missing intervals to be unmatched by setting this to FALSE, and you can force them to be propagated by setting this to NA.

Value

A logical vector the same size as needles.

Allen's Interval Algebra

The interval algebra developed by James Allen serves as a basis and inspiration for iv_locate_overlaps(), iv_locate_precedes(), and iv_locate_follows(). The original algebra is composed of 13 relations which have the following properties:

- Distinct: No pair of intervals can be related by more than one type.
- Exhaustive: All pairs of intervals are described by one of the types.
- Qualitative: No numeric intervals are considered. The relationships are computed by purely qualitative means.

Take the notation that x and y represent two intervals. Now assume that x can be represented as [x_s, x_e), where x_s is the start of the interval and x_e is the end of it. Additionally, assume that x_s < x_e. With this notation, the 13 relations are as follows:

- *Precedes*:
  x_e < y_s
- *Preceded-by*:
  x_s > y_e
- *Meets*:
  x_e == y_s
- *Met-by*:
  x_s == y_e
- *Overlaps*:
  (x_s < y_s) & (x_e > y_s) & (x_e < y_e)
- *Overlapped-by*:
  (x_e > y_e) & (x_s < y_e) & (x_s > y_s)

- *Starts*:
  `(x_s == y_s) & (x_e < y_e)`
- *Started-by*:
  `(x_s == y_s) & (x_e > y_e)`
- *Finishes*:
  `(x_s > y_s) & (x_e == y_e)`
- *Finished-by*:
  `(x_s < y_s) & (x_e == y_e)`
- *During*:
  `(x_s > y_s) & (x_e < y_e)`
- *Contains*:
  `(x_s < y_s) & (x_e > y_e)`
- *Equals*:
  `(x_s == y_s) & (x_e == y_e)`

Note that when `missing = "equals"`, missing intervals will only match the `type = "equals"` relation. This ensures that the distinct property of the algebra is maintained.

Connection to other functions:

Note that some of the above relations are fairly restrictive. For example, `"overlaps"` only detects cases where `x` straddles `y_s`. It does not consider the case where `x` and `y` are equal to be an overlap (as this is `"equals"`) nor does it consider when `x` straddles `y_e` to be an overlap (as this is `"overlapped-by"`). This makes the relations extremely useful from a theoretical perspective, because they can be combined without fear of duplicating relations, but they don't match our typical expectations for what an "overlap" is.

`iv_locate_overlaps()`, `iv_locate_precedes()`, and `iv_locate_follows()` use more intuitive types that aren't distinct, but typically match your expectations better. They can each be expressed in terms of Allen's relations:

- `iv_locate_overlaps()`:
  - `"any"`:
    `overlaps | overlapped-by | starts | started-by | finishes | finished-by | during | contains | equals`
  - `"contains"`:
    `contains | started-by | finished-by | equals`
  - `"within"`:
    `during | starts | finishes | equals`
  - `"starts"`:
    `starts | started-by | equals`
  - `"ends"`:
    `finishes | finished-by | equals`
  - `"equals"`:
    `equals`
- `iv_locate_precedes()`:
  `precedes | meets`
- `iv_locate_follows()`:
  `preceded-by | met-by`

See Also

Locating relationships

Locating relations from Allen's Interval Algebra

Detecting relations from Allen's Interval Algebra pairwise

Examples

```
x <- iv(1, 3)
y <- iv(3, 4)

# `"precedes"` is strict, and doesn't let the endpoints match
iv_relates(x, y, type = "precedes")

# Since that is what `"meets"` represents
iv_relates(x, y, type = "meets")

# `"overlaps"` is a very specific type of overlap where an interval in
# `needles` straddles the start of an interval in `haystack`
x <- iv_pairs(c(1, 4), c(1, 3), c(0, 3), c(2, 5))
y <- iv(1, 4)

# It doesn't match equality, or when the starts match, or when the end
# of the interval in `haystack` is straddled instead
iv_relates(x, y, type = "overlaps")
```

`allen-relation-detect-pairwise`

Pairwise detect relations from Allen's Interval Algebra

Description

`iv_pairwise_relates()` is similar to `iv_pairwise_overlaps()`, but it detects a specific set of relations developed by James Allen in the paper: Maintaining Knowledge about Temporal Intervals.

Usage

```
iv_pairwise_relates(x, y, ..., type)
```

Arguments

`x, y`	`[iv]` A pair of interval vectors. These will be recycled against each other and cast to the same type.
`...`	These dots are for future extensions and must be empty.
`type`	`[character(1)]` The type of relationship to find. See the Allen's Interval Algebra section for a complete description of each type. One of:

- "precedes"
- "preceded-by"
- "meets"
- "met-by"
- "overlaps"
- "overlapped-by"
- "starts"
- "started-by"
- "during"
- "contains"
- "finishes"
- "finished-by"
- "equals"

Value

A logical vector the same size as the common size of x and y.

Allen's Interval Algebra

The interval algebra developed by James Allen serves as a basis and inspiration for iv_locate_overlaps(), iv_locate_precedes(), and iv_locate_follows(). The original algebra is composed of 13 relations which have the following properties:

- Distinct: No pair of intervals can be related by more than one type.
- Exhaustive: All pairs of intervals are described by one of the types.
- Qualitative: No numeric intervals are considered. The relationships are computed by purely qualitative means.

Take the notation that x and y represent two intervals. Now assume that x can be represented as [x_s, x_e), where x_s is the start of the interval and x_e is the end of it. Additionally, assume that x_s < x_e. With this notation, the 13 relations are as follows:

- *Precedes*:
  x_e < y_s
- *Preceded-by*:
  x_s > y_e
- *Meets*:
  x_e == y_s
- *Met-by*:
  x_s == y_e
- *Overlaps*:
  (x_s < y_s) & (x_e > y_s) & (x_e < y_e)
- *Overlapped-by*:
  (x_e > y_e) & (x_s < y_e) & (x_s > y_s)

- *Starts*:
  `(x_s == y_s) & (x_e < y_e)`
- *Started-by*:
  `(x_s == y_s) & (x_e > y_e)`
- *Finishes*:
  `(x_s > y_s) & (x_e == y_e)`
- *Finished-by*:
  `(x_s < y_s) & (x_e == y_e)`
- *During*:
  `(x_s > y_s) & (x_e < y_e)`
- *Contains*:
  `(x_s < y_s) & (x_e > y_e)`
- *Equals*:
  `(x_s == y_s) & (x_e == y_e)`

Note that when `missing = "equals"`, missing intervals will only match the `type = "equals"` relation. This ensures that the distinct property of the algebra is maintained.

Connection to other functions:

Note that some of the above relations are fairly restrictive. For example, `"overlaps"` only detects cases where `x` straddles `y_s`. It does not consider the case where `x` and `y` are equal to be an overlap (as this is `"equals"`) nor does it consider when `x` straddles `y_e` to be an overlap (as this is `"overlapped-by"`). This makes the relations extremely useful from a theoretical perspective, because they can be combined without fear of duplicating relations, but they don't match our typical expectations for what an "overlap" is.

`iv_locate_overlaps()`, `iv_locate_precedes()`, and `iv_locate_follows()` use more intuitive types that aren't distinct, but typically match your expectations better. They can each be expressed in terms of Allen's relations:

- `iv_locate_overlaps()`:
  - `"any"`:
    `overlaps | overlapped-by | starts | started-by | finishes | finished-by | during | contains | equals`
  - `"contains"`:
    `contains | started-by | finished-by | equals`
  - `"within"`:
    `during | starts | finishes | equals`
  - `"starts"`:
    `starts | started-by | equals`
  - `"ends"`:
    `finishes | finished-by | equals`
  - `"equals"`:
    `equals`
- `iv_locate_precedes()`:
  `precedes | meets`
- `iv_locate_follows()`:
  `preceded-by | met-by`

See Also

Locating relationships

Locating relations from Allen's Interval Algebra

Detecting relations from Allen's Interval Algebra

Examples

```
x <- iv_pairs(c(1, 3), c(3, 5))
y <- iv_pairs(c(3, 4), c(6, 7))

# `"precedes"` is strict, and doesn't let the endpoints match
iv_pairwise_relates(x, y, type = "precedes")

# Since that is what `"meets"` represents
iv_pairwise_relates(x, y, type = "meets")

# `"during"` only matches when `x` is completely contained in `y`, and
# doesn't allow any endpoints to match
x <- iv_pairs(c(1, 3), c(4, 5), c(8, 9))
y <- iv_pairs(c(1, 4), c(3, 8), c(8, 9))

iv_pairwise_relates(x, y, type = "during")
```

---

allen-relation-locate *Locate relations from Allen's Interval Algebra*

Description

`iv_locate_relates()` is similar to `iv_locate_overlaps()`, but it locates a specific set of relations developed by James Allen in the paper: Maintaining Knowledge about Temporal Intervals.

Usage

```
iv_locate_relates(
  needles,
  haystack,
  ...,
  type,
  missing = "equals",
  no_match = NA_integer_,
  remaining = "drop",
  multiple = "all",
  relationship = "none"
)
```

Arguments

needles, haystack

[iv]

Interval vectors used for relation matching.

- Each element of needles represents the interval to search for.
- haystack represents the intervals to search in.

Prior to comparison, needles and haystack are coerced to the same type.

...
These dots are for future extensions and must be empty.

type
[character(1)]

The type of relationship to find. See the Allen's Interval Algebra section for a complete description of each type. One of:

- "precedes"
- "preceded-by"
- "meets"
- "met-by"
- "overlaps"
- "overlapped-by"
- "starts"
- "started-by"
- "during"
- "contains"
- "finishes"
- "finished-by"
- "equals"

missing
[integer(1) / "equals" / "drop" / "error"]

Handling of missing intervals in needles.

- "equals" considers missing intervals in needles as exactly equal to missing intervals in haystack when determining if there is a matching relationship between them.
- "drop" drops missing intervals in needles from the result.
- "error" throws an error if any intervals in needles are missing.
- If a single integer is provided, this represents the value returned in the haystack column for intervals in needles that are missing.

no_match
Handling of needles without a match.

- "drop" drops needles with zero matches from the result.
- "error" throws an error if any needles have zero matches.
- If a single integer is provided, this represents the value returned in the haystack column for values of needles that have zero matches. The default represents an unmatched needle with NA.

remaining
Handling of haystack values that needles never matched.

- "drop" drops remaining haystack values from the result. Typically, this is the desired behavior if you only care when needles has a match.

- `"error"` throws an error if there are any remaining `haystack` values.
- If a single integer is provided (often `NA`), this represents the value returned in the `needles` column for the remaining `haystack` values that `needles` never matched. Remaining `haystack` values are always returned at the end of the result.

`multiple` Handling of `needles` with multiple matches. For each needle:

- `"all"` returns all matches detected in `haystack`.
- `"any"` returns any match detected in `haystack` with no guarantees on which match will be returned. It is often faster than `"first"` and `"last"` if you just need to detect if there is at least one match.
- `"first"` returns the first match detected in `haystack`.
- `"last"` returns the last match detected in `haystack`.

`relationship` Handling of the expected relationship between `needles` and `haystack`. If the expectations chosen from the list below are invalidated, an error is thrown.

- `"none"` doesn't perform any relationship checks.
- `"one-to-one"` expects:
  - Each value in `needles` matches at most 1 value in `haystack`.
  - Each value in `haystack` matches at most 1 value in `needles`.
- `"one-to-many"` expects:
  - Each value in `needles` matches any number of values in `haystack`.
  - Each value in `haystack` matches at most 1 value in `needles`.
- `"many-to-one"` expects:
  - Each value in `needles` matches at most 1 value in `haystack`.
  - Each value in `haystack` matches any number of values in `needles`.
- `"many-to-many"` expects:
  - Each value in `needles` matches any number of values in `haystack`.
  - Each value in `haystack` matches any number of values in `needles`.

  This performs no checks, and is identical to `"none"`, but is provided to allow you to be explicit about this relationship if you know it exists.
- `"warn-many-to-many"` doesn't assume there is any known relationship, but will warn if `needles` and `haystack` have a many-to-many relationship (which is typically unexpected), encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying `"many-to-many"`.

`relationship` is applied after `filter` and `multiple` to allow potential multiple matches to be filtered out first.

`relationship` doesn't handle cases where there are zero matches. For that, see `no_match` and `remaining`.

Value

A data frame containing two integer columns named `needles` and `haystack`.

Allen's Interval Algebra

The interval algebra developed by James Allen serves as a basis and inspiration for `iv_locate_overlaps()`, `iv_locate_precedes()`, and `iv_locate_follows()`. The original algebra is composed of 13 relations which have the following properties:

- Distinct: No pair of intervals can be related by more than one `type`.
- Exhaustive: All pairs of intervals are described by one of the `types`.
- Qualitative: No numeric intervals are considered. The relationships are computed by purely qualitative means.

Take the notation that `x` and `y` represent two intervals. Now assume that `x` can be represented as `[x_s, x_e)`, where `x_s` is the start of the interval and `x_e` is the end of it. Additionally, assume that `x_s < x_e`. With this notation, the 13 relations are as follows:

- *Precedes*:
  `x_e < y_s`
- *Preceded-by*:
  `x_s > y_e`
- *Meets*:
  `x_e == y_s`
- *Met-by*:
  `x_s == y_e`
- *Overlaps*:
  `(x_s < y_s) & (x_e > y_s) & (x_e < y_e)`
- *Overlapped-by*:
  `(x_e > y_e) & (x_s < y_e) & (x_s > y_s)`
- *Starts*:
  `(x_s == y_s) & (x_e < y_e)`
- *Started-by*:
  `(x_s == y_s) & (x_e > y_e)`
- *Finishes*:
  `(x_s > y_s) & (x_e == y_e)`
- *Finished-by*:
  `(x_s < y_s) & (x_e == y_e)`
- *During*:
  `(x_s > y_s) & (x_e < y_e)`
- *Contains*:
  `(x_s < y_s) & (x_e > y_e)`
- *Equals*:
  `(x_s == y_s) & (x_e == y_e)`

Note that when `missing = "equals"`, missing intervals will only match the `type = "equals"` relation. This ensures that the distinct property of the algebra is maintained.

Connection to other functions:

Note that some of the above relations are fairly restrictive. For example, `"overlaps"` only detects cases where `x` straddles `y_s`. It does not consider the case where `x` and `y` are equal to be an overlap (as this is `"equals"`) nor does it consider when `x` straddles `y_e` to be an overlap (as this is `"overlapped-by"`). This makes the relations extremely useful from a theoretical perspective, because they can be combined without fear of duplicating relations, but they don't match our typical expectations for what an "overlap" is.

`iv_locate_overlaps()`, `iv_locate_precedes()`, and `iv_locate_follows()` use more intuitive types that aren't distinct, but typically match your expectations better. They can each be expressed in terms of Allen's relations:

- `iv_locate_overlaps()`:
  - `"any"`:
    `overlaps | overlapped-by | starts | started-by | finishes | finished-by | during | contains | equals`
  - `"contains"`:
    `contains | started-by | finished-by | equals`
  - `"within"`:
    `during | starts | finishes | equals`
  - `"starts"`:
    `starts | started-by | equals`
  - `"ends"`:
    `finishes | finished-by | equals`
  - `"equals"`:
    `equals`
- `iv_locate_precedes()`:
  `precedes | meets`
- `iv_locate_follows()`:
  `preceded-by | met-by`

References

Allen, James F. (26 November 1983). "Maintaining knowledge about temporal intervals". Communications of the ACM. 26 (11): 832–843.

See Also

Locating relationships

Detecting relations from Allen's Interval Algebra

Detecting relations from Allen's Interval Algebra pairwise

Examples

```
x <- iv(1, 3)
y <- iv(3, 4)

# `"precedes"` is strict, and doesn't let the endpoints match
iv_locate_relates(x, y, type = "precedes")
```

```
# Since that is what `"meets"` represents
iv_locate_relates(x, y, type = "meets")

# `"overlaps"` is a very specific type of overlap where an interval in
# `needles` straddles the start of an interval in `haystack`
x <- iv_pairs(c(1, 4), c(1, 3), c(0, 3), c(2, 5))
y <- iv(1, 4)

# It doesn't match equality, or when the starts match, or when the end
# of the interval in `haystack` is straddled instead
iv_locate_relates(x, y, type = "overlaps")
```

`is_iv` *Is* x *an iv?*

Description

`is_iv()` tests if x is an iv object.

Usage

```
is_iv(x)
```

Arguments

x	`[object]`
	An object.

Value

A single `TRUE` or `FALSE`.

Examples

```
is_iv(1)
is_iv(new_iv(1, 2))
```

iv *Create an interval vector*

Description

- `iv()` creates an interval vector from `start` and `end` vectors. This is how you will typically create interval vectors, and is often used with columns in a data frame.
- `iv_pairs()` creates an interval vector from *pairs*. This is often useful for interactive testing, as it provides a more intuitive interface for creating small interval vectors. It should generally not be used on a large scale because it can be slow.

Intervals:

Interval vectors are *right-open*, i.e. `[start, end)`. This means that `start < end` is a requirement to generate an interval vector. In particular, empty intervals with `start == end` are not allowed.

Right-open intervals tend to be the most practically useful. For example, `[2019-01-01 00:00:00, 2019-01-02 00:00:00)` nicely encapsulates all times on `2019-01-01`. With closed intervals, you'd have to attempt to specify this as `2019-01-01 23:59:59`, which is inconvenient and inaccurate, as it doesn't capture fractional seconds.

Right-open intervals also have the extremely nice technical property that they create a closed algebra. Concretely, the complement of a vector of right-open intervals and the union, intersection, or difference of two vectors of right-open intervals will always result in another vector of right-open intervals.

Missing intervals:

When creating interval vectors with `iv()`, if either bound is incomplete, then both bounds are set to their missing value.

Usage

```
iv(start, end, ..., ptype = NULL, size = NULL)

iv_pairs(..., ptype = NULL)
```

Arguments

`start`, `end`	`[vector]` A pair of vectors to represent the bounds of the intervals. To be a valid interval vector, `start` must be strictly less than `end`. If either `start` or `end` are incomplete / missing, then both bounds will be coerced to missing values. `start` and `end` are recycled against each other and are cast to the same type.
`...`	For `iv_pairs()`: `[vector pairs]` Vectors of size 2 representing intervals to include in the result. All inputs will be cast to the same type. For `iv()`: These dots are for future extensions and must be empty.

ptype [vector(0) / NULL]

A prototype to force for the inner type of the resulting iv. If NULL, this defaults to the common type of the inputs.

size [integer(1) / NULL]

A size to force for the resulting iv. If NULL, this defaults to the common size of the inputs.

Value

An iv.

Examples

```
library(dplyr, warn.conflicts = FALSE)

set.seed(123)

x <- tibble(
  start = as.Date("2019-01-01") + 1:5,
  end = start + sample(1:10, length(start), replace = TRUE)
)

# Typically you'll use `iv()` with columns of a data frame
mutate(x, iv = iv(start, end), .keep = "unused")

# `iv_pairs()` is useful for generating interval vectors interactively
iv_pairs(c(1, 5), c(2, 3), c(6, 10))
```

---

iv-accessors *Access the start or end of an interval vector*

Description

- iv_start() accesses the start of an interval vector.
- iv_end() accesses the end of an interval vector.

Usage

```
iv_start(x)

iv_end(x)
```

Arguments

x [iv]

An interval vector.

Value

The start or end of `x`.

Examples

```
x <- new_iv(1, 2)

iv_start(x)
iv_end(x)
```

`iv-containers` *Containers*

Description

This family of functions revolves around computing interval *containers*. A container is defined as the widest interval that isn't contained by any other interval.

- `iv_containers()` returns all of the containers found within `x`.
- `iv_identify_containers()` identifies the containers that each interval in `x` falls in. It replaces `x` with a list of the same size where each element of the list contains the containers that the corresponding interval in `x` falls in. This is particularly useful alongside `tidyr::unnest()`.
- `iv_identify_container()` is similar in spirit to `iv_identify_containers()`, but is useful when you suspect that each interval in `x` is contained within exactly 1 container. It replaces `x` with an iv of the same size where each interval is the container that the corresponding interval in `x` falls in. If any interval falls in more than one container, an error is thrown.
- `iv_locate_containers()` returns a two column data frame with a `key` column containing the result of `iv_containers()` and a `loc` list-column containing integer vectors that map each interval in `x` to the container that it falls in.

Usage

```
iv_containers(x)

iv_identify_containers(x)

iv_identify_container(x)

iv_locate_containers(x)
```

Arguments

`x`	`[iv]` An interval vector.

Value

- For `iv_containers()`, an iv with the same type as `x`.
- For `iv_identify_containers()`, a list-of containing ivs with the same size as `x`.
- For `iv_identify_container()`, an iv with the same type as `x`.
- For `iv_locate_containers()`, a two column data frame with a `key` column containing the result of `iv_containers()` and a `loc` list-column containing integer vectors.

Examples

```
library(dplyr, warn.conflicts = FALSE)
library(tidyr)

x <- iv_pairs(
  c(4, 6),
  c(1, 5),
  c(2, 3),
  c(NA, NA),
  c(NA, NA),
  c(9, 12),
  c(9, 14)
)
x

# Containers are intervals which aren't contained in any other interval.
# They are always returned in ascending order.
# If any missing intervals are present, a single one is retained.
iv_containers(x)

# `iv_identify_container()` is useful alongside `group_by()` and
# `summarize()` if you know that each interval is contained within exactly
# 1 container
df <- tibble(x = x)
df <- mutate(df, container = iv_identify_container(x))
df

df %>%
  group_by(container) %>%
  summarize(n = n())

# If any interval is contained within multiple containers,
# then you can't use `iv_identify_container()`
y <- c(x, iv_pairs(c(0, 3), c(8, 13)))
y

try(iv_identify_container(y))

# Instead, use `iv_identify_containers()` to identify every container
# that each interval falls in
df <- tibble(y = y, container = iv_identify_containers(y))
df
```

```
# You can use `tidyr::unchop()` to see the containers that each interval
# falls in
df %>%
  mutate(row = row_number(), .before = 1) %>%
  unchop(container)

# A more programmatic interface to `iv_identify_containers()` is
# `iv_locate_containers()`, which returns the containers you get from
# `iv_containers()` alongside the locations in the input that they contain.
iv_locate_containers(y)
```

`iv-genericity` *Proxy and restore*

Description

- `iv_proxy()` is an S3 generic which allows you to write S3 methods for iv extension types to ensure that they are treated like iv objects. The input will be your iv extension object, `x`, and the return value should be an iv object.
- `iv_restore()` is an S3 generic that dispatches off `to` that allows you to restore a proxied iv extension type back to its original type. The inputs will be a bare iv object, `x`, and your original iv extension object, `to`, and the return value should correspond to `x` restored to the type of `to`, if possible.

You typically *don't* need to create an `iv_proxy()` method if your class directly extends iv through the `class` argument of `new_iv()`. You only need to implement this if your class has a different structure than a typical iv object. In particular, if `vctrs::field(x, "start")` and `vctrs::field(x, "end")` don't return the `start` and `end` of the interval vector respectively, then you probably need an `iv_proxy()` method.

You typically *do* need an `iv_restore()` method for custom iv extensions. If your class is simple, then you can generally just call your constructor, like `new_my_iv()`, to restore the class and any additional attributes that might be required. If your class doesn't use `new_iv()`, then an `iv_restore()` method is mandatory, as this is one of the ways that ivs detects that your class is compatible with ivs.

This system allows you to use any `iv_*()` function on your iv extension object without having to define S3 methods for all of them.

Note that the default method for `iv_proxy()` returns its input unchanged, even if it isn't an iv. Each `iv_*()` function does separate checking to ensure that the proxy is a valid iv, or implements an alternate behavior if no proxy method is implemented. In contrast, `iv_restore()` will error if a method for `to` isn't registered.

Usage

```
iv_proxy(x, ...)

iv_restore(x, to, ...)
```

Arguments

x	[vector] A vector.
...	These dots are for future extensions and must be empty.
to	[vector] The original vector to restore to.

Value

- `iv_proxy()` should return an iv object for further manipulation.
- `iv_restore()` should return an object of type `to`, if possible. In some cases, it may be required to fall back to returning an iv object.

Examples

```
if (FALSE) {
# Registering S3 methods outside of a package doesn't always work quite
# right (like on the pkgdown site), so this code should only be run by a
# user reading the manual. If that is you, fear not! It should run just fine
# in your console.

library(vctrs)

new_nested_iv <- function(iv) {
  fields <- list(iv = iv)
  new_rcrd(fields, class = "nested_iv")
}

format.nested_iv <- function(x, ...) {
  format(field(x, "iv"))
}

iv_proxy.nested_iv <- function(x, ...) {
  field(x, "iv")
}

iv_restore.nested_iv <- function(x, to, ...) {
  new_nested_iv(x)
}

iv <- new_iv(c(1, 5), c(2, 7))

x <- new_nested_iv(iv)
x

# Proxies, then accesses the `start` field
iv_start(x)

# Proxies, computes the complement to generate an iv,
# then restores to the original type
```

```
iv_set_complement(x)

}
```

`iv-groups` *Group overlapping intervals*

Description

This family of functions revolves around grouping overlapping intervals within a single iv. When multiple overlapping intervals are grouped together they result in a wider interval containing the smallest `iv_start()` and the largest `iv_end()` of the overlaps.

- `iv_groups()` merges all overlapping intervals found within `x`. The resulting intervals are known as the "groups" of `x`.
- `iv_identify_group()` identifies the group that the current interval of `x` falls in. This is particularly useful alongside `dplyr::group_by()`.
- `iv_locate_groups()` returns a two column data frame with a `key` column containing the result of `iv_groups()` and a `loc` list-column containing integer vectors that map each interval in `x` to the group that it falls in.

Optionally, you can choose *not* to group abutting intervals together with `abutting = FALSE`, which can be useful if you'd like to retain those boundaries.

Minimal interval vectors:

`iv_groups()` is particularly useful because it can generate a *minimal* interval vector, which covers the range of an interval vector in the most compact form possible. In particular, a minimal interval vector:

- Has no overlapping intervals
- Has no abutting intervals
- Is ordered on both `start` and `end`

A minimal interval vector is allowed to have a single missing interval, which is located at the end of the vector.

Usage

```
iv_groups(x, ..., abutting = TRUE)

iv_identify_group(x, ..., abutting = TRUE)

iv_locate_groups(x, ..., abutting = TRUE)
```

Arguments

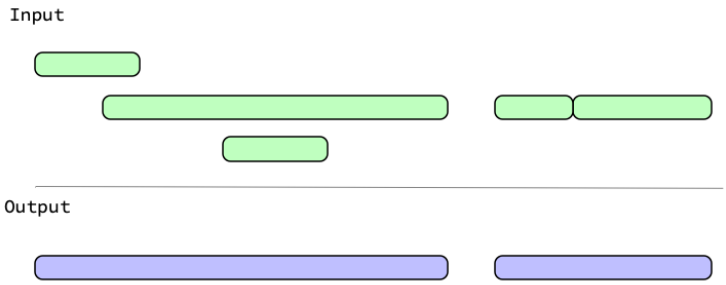
x	[iv] An interval vector.
...	These dots are for future extensions and must be empty.
abutting	[TRUE / FALSE] Should abutting intervals be grouped together? If TRUE, [a, b) and [b, c) will merge as [a, c). If FALSE, they will be kept separate. To be a minimal interval vector, all abutting intervals must be grouped together.

Value

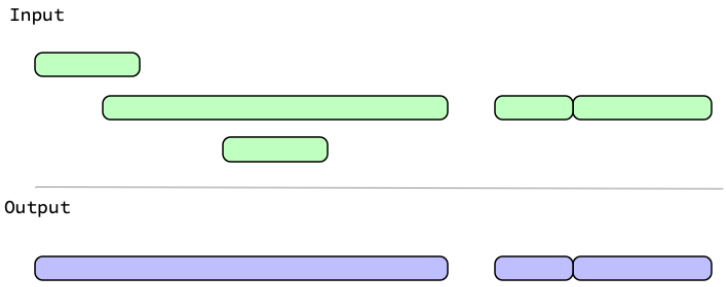
- For iv_groups(), an iv with the same type as x.
- For iv_identify_group(), an iv with the same type and size as x.
- For iv_locate_groups(), a two column data frame with a key column containing the result of iv_groups() and a loc list-column containing integer vectors.

Graphical Representation

Graphically, generating groups looks like:



With abutting = FALSE, intervals that touch aren't grouped:



Examples

```
library(dplyr, warn.conflicts = FALSE)

x <- iv_pairs(
  c(1, 5),
  c(2, 3),
  c(NA, NA),
  c(5, 6),
  c(NA, NA),
  c(9, 12),
  c(11, 14)
)
x

# Grouping removes all redundancy while still covering the full range
# of values that were originally represented. If any missing intervals
# are present, a single one is retained.
iv_groups(x)

# Abutting intervals are typically grouped together, but you can choose not
# to group them if you want to retain those boundaries
iv_groups(x, abutting = FALSE)

# `iv_identify_group()` is useful alongside `group_by()` and `summarize()`
df <- tibble(x = x)
df <- mutate(df, u = iv_identify_group(x))
df

df %>%
  group_by(u) %>%
  summarize(n = n())

# The real workhorse here is `iv_locate_groups()`, which returns
# the groups and information on which observations in `x` fall in which
# group
iv_locate_groups(x)
```

---

iv-set-pairwise *Pairwise set operations*

Description

This family of functions performs *pairwise* set operations on two ivs. Pairwise refers to the fact that the i-th interval of `x` is going to be compared against the i-th interval of `y`. This is in contrast to their counterparts, like `iv_set_union()`, which treat the entire vector of `x` as a single set to be compared against all of `y`.

The descriptions of these operations are the same as their non-pairwise counterparts, but the ones here also have a number of restrictions due to the fact that each must return an output that is the same size as its inputs:

- For `iv_pairwise_set_complement()`, `x[i]` and `y[i]` can't overlap or abut, as this would generate an empty complement.
- For `iv_pairwise_set_union()`, `x[i]` and `y[i]` can't be separated by a gap. Use `iv_pairwise_span()` if you want to force gaps to be filled anyways.
- For `iv_pairwise_set_intersect()`, `x[i]` and `y[i]` must overlap, otherwise an empty interval would be generated.
- For `iv_pairwise_set_difference()`, `x[i]` can't be completely contained within `y[i]`, as that would generate an empty interval. Additionally, `y[i]` can't be completely contained within `x[i]`, as that would result in two distinct intervals for a single observation.
- For `iv_pairwise_set_symmetric_difference()`, `x[i]` and `y[i]` must share exactly one endpoint, otherwise an empty interval or two distinct intervals would be generated.

Usage

```
iv_pairwise_set_complement(x, y)

iv_pairwise_set_union(x, y)

iv_pairwise_set_intersect(x, y)

iv_pairwise_set_difference(x, y)

iv_pairwise_set_symmetric_difference(x, y)
```

Arguments

`x, y`	`[iv]` A pair of interval vectors. These will be cast to the same type, and recycled against each other.

Value

An iv the same size and type as `x` and `y`.

See Also

The non-pairwise versions of these functions, such as `iv_set_union()`.

Examples

```
x <- iv_pairs(c(1, 3), c(6, 8))
y <- iv_pairs(c(5, 7), c(2, 3))

iv_pairwise_set_complement(x, y)

z <- iv_pairs(c(2, 5), c(4, 7))

iv_pairwise_set_union(x, z)
```

```
# Can't take the union when there are gaps
try(iv_pairwise_set_union(x, y))

# But you can force a union across gaps with `iv_pairwise_span()`
iv_pairwise_span(x, y)

iv_pairwise_set_intersect(x, z)

# Can't take an intersection of non-overlapping intervals
try(iv_pairwise_set_intersect(x, y))

iv_pairwise_set_difference(x, z)

# The pairwise symmetric difference function is fairly strict,
# and is only well defined when exactly one of the interval endpoints match
w <- iv_pairs(c(1, 6), c(7, 8))
iv_pairwise_set_symmetric_difference(x, w)
```

`iv-sets` *Set operations*

Description

This family of functions treats ivs as sets. They always compute the minimal iv of each input and return a minimal iv.

- `iv_set_complement()` takes the complement of the intervals in an iv. By default, the minimum and maximum of the inputs define the bounds to take the complement over, but this can be adjusted with `lower` and `upper`. Missing intervals are always dropped in the complement.
- `iv_set_union()` answers the question, "Which intervals are in `x` or `y`?" It is equivalent to combining the two vectors together and then calling `iv_groups()`.
- `iv_set_intersect()` answers the question, "Which intervals are in `x` and `y`?"
- `iv_set_difference()` answers the question, "Which intervals are in `x` but not `y`?" Note that this is an asymmetrical difference.
- `iv_set_symmetric_difference()` answers the question, "Which intervals are in `x` or `y` but not both?"

Usage

```
iv_set_complement(x, ..., lower = NULL, upper = NULL)

iv_set_union(x, y)

iv_set_intersect(x, y)

iv_set_difference(x, y)

iv_set_symmetric_difference(x, y)
```

Arguments

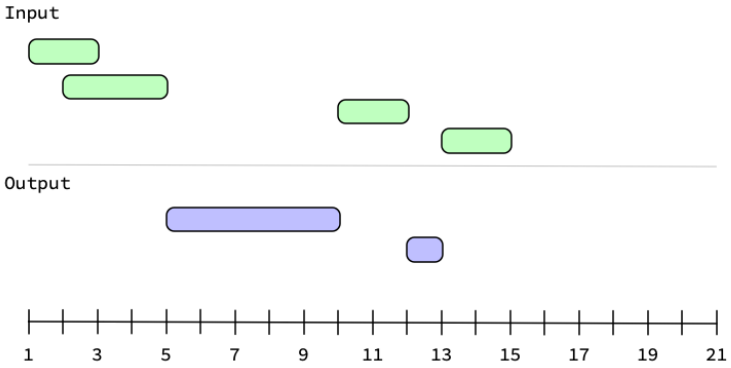
`x`	`[iv]` An interval vector.
`...`	These dots are for future extensions and must be empty.
`lower, upper`	`[vector(1) / NULL]` Bounds for the universe over which to compute the complement. These should have the same type as the element type of the interval vector. It is often useful to expand the universe to, say, `-Inf` to `Inf`.
`y`	`[iv]` An interval vector.

Value

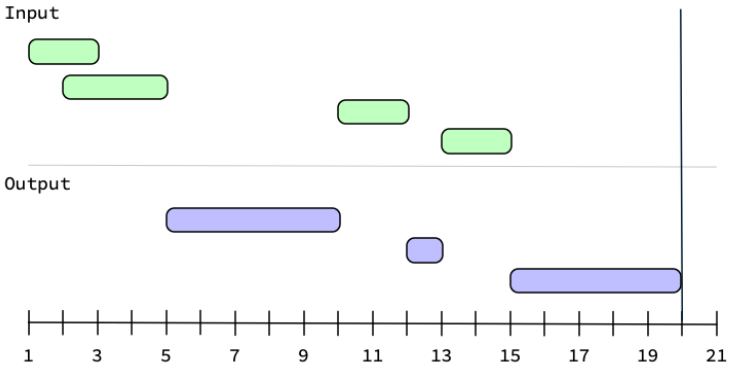
- For `iv_set_complement()`, a vector of the same type as `x` containing the complement.
- For all other set operations, a vector of the same type as the common type of `x` and `y` containing the result.

Graphical Representation

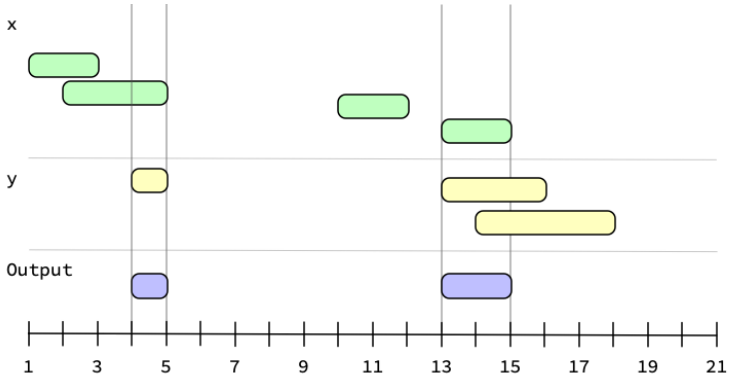
Graphically, generating the complement looks like:



If you were to set `upper = 20` with these intervals, then you'd get one more interval in the complement.



Generating the intersection between two ivs looks like:



See Also

The *pairwise* versions of these functions, such as `iv_pairwise_set_union()`.

Examples

```
x <- iv_pairs(
  c(10, 12),
  c(0, 5),
  c(NA, NA),
  c(3, 6),
  c(-5, -2),
  c(NA, NA)
)
x

y <- iv_pairs(
  c(2, 7),
  c(NA, NA),
  c(-3, -1),
  c(14, 15)
)
y

# Complement contains any values from `[-5, 12)` that aren't represented
# in these intervals. Missing intervals are dropped.
iv_set_complement(x)

# Expand out the "universe" of possible values
iv_set_complement(x, lower = -Inf)
iv_set_complement(x, lower = -Inf, upper = Inf)

# Which intervals are in x or y?
iv_set_union(x, y)

# Which intervals are in x and y?
iv_set_intersect(x, y)
```

```
# Which intervals are in x but not y?
iv_set_difference(x, y)

# Which intervals are in y but not x?
iv_set_difference(y, x)

# Missing intervals in x are kept if there aren't missing intervals in y
iv_set_difference(x, iv(1, 2))

# Which intervals are in x or y but not both?
iv_set_symmetric_difference(x, y)

# Missing intervals will be kept if they only appear on one side
iv_set_symmetric_difference(x, iv(1, 2))
iv_set_symmetric_difference(iv(1, 2), x)
```

`iv-splits` *Splits*

Description

This family of functions revolves around splitting an iv on its endpoints, which results in a new iv that is entirely disjoint (i.e. non-overlapping). The intervals in the resulting iv are known as "splits".

- `iv_splits()` computes the disjoint splits for `x`.
- `iv_identify_splits()` identifies the splits that correspond to each interval in `x`. It replaces `x` with a list of the same size where each element of the list contains the splits that the corresponding interval in `x` overlaps. This is particularly useful alongside `tidyr::unnest()`.
- `iv_locate_splits()` returns a two column data frame with a `key` column containing the result of `iv_splits()` and a `loc` list-column containing integer vectors that map each interval in `x` to the splits that it overlaps.

Usage

```
iv_splits(x, ..., on = NULL)

iv_identify_splits(x, ..., on = NULL)

iv_locate_splits(x, ..., on = NULL)
```

Arguments

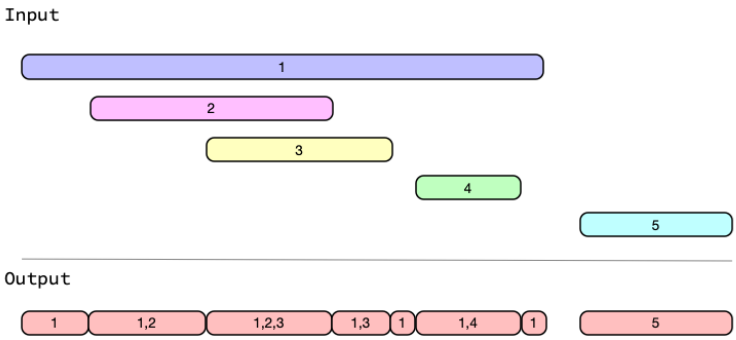
`x`	`[iv]` An interval vector.
`...`	These dots are for future extensions and must be empty.
`on`	`[vector / NULL]` An optional vector of additional values to split on. This should have the same type as `iv_start(x)`.

Value

- For `iv_splits()`, an iv with the same type as `x`.
- For `iv_identify_splits()`, a list-of containing ivs with the same size as `x`.
- For `iv_locate_splits()`, a two column data frame with a `key` column of the same type as `x` and `loc` list-column containing integer vectors.

Graphical Representation

Graphically, generating splits looks like:



Examples

```
library(tidyr)
library(dplyr)

# Guests to a party and their arrival/departure times
guests <- tibble(
  arrive = as.POSIXct(
    c("2008-05-20 19:30:00", "2008-05-20 20:10:00", "2008-05-20 22:15:00"),
    tz = "UTC"
  ),
  depart = as.POSIXct(
    c("2008-05-20 23:00:00", "2008-05-21 00:00:00", "2008-05-21 00:30:00"),
    tz = "UTC"
  ),
  name = list(
    c("Mary", "Harry"),
    c("Diana", "Susan"),
    "Peter"
  )
)

guests <- unnest(guests, name) %>%
  mutate(iv = iv(arrive, depart), .keep = "unused")

guests
```

```
# You can determine the disjoint intervals at which people
# arrived/departed with `iv_splits()`
iv_splits(guests$iv)

# Say you'd like to determine who was at the party at any given time
# throughout the night
guests <- mutate(guests, splits = iv_identify_splits(iv))
guests

# Unnest the splits to generate disjoint intervals for each guest
guests <- guests %>%
  unnest(splits) %>%
  select(name, splits)

guests

# Tabulate who was there at any given time
guests %>%
  summarise(n = n(), who = list(name), .by = splits)

# ------------------------------------------------------------------------

x <- iv_pairs(c(1, 5), c(4, 9), c(12, 15))
x

# You can provide additional singular values to split on with `on`
iv_splits(x, on = c(2, 13))
```

`iv_align` *Align after locating relationships*

Description

`iv_align()` will align/join `needles` and `haystack` together using a data frame of `locations`. These `locations` are intended to be the output of one of: `iv_locate_overlaps()`, `iv_locate_precedes()`, `iv_locate_follows()`, `iv_locate_relates()`, or `iv_locate_between()`.

This is mainly a convenience function that slices both `needles` and `haystack` according to those `locations`, and then stores the result in a new two column data frame.

Usage

```
iv_align(needles, haystack, ..., locations)
```

Arguments

`needles`, `haystack`	`[vector]` Two vectors to align.
`...`	These dots are for future extensions and must be empty.

locations [two-column data frame]

The data frame of locations returned from one of iv_locate_overlaps(), iv_locate_precedes(), iv_locate_follows(), iv_locate_relates(), or iv_locate_between().

Value

A two column data frame with a $needles column containing the sliced version of needles and a $haystack column containing the sliced version of haystack.

Examples

```
needles <- iv_pairs(c(1, 5), c(3, 7), c(10, 12))
haystack <- iv_pairs(c(0, 2), c(4, 6))

locations <- iv_locate_overlaps(needles, haystack)
iv_align(needles, haystack, locations = locations)

locations <- iv_locate_overlaps(needles, haystack, no_match = "drop")
iv_align(needles, haystack, locations = locations)

needles <- c(1, 15, 4, 11)
haystack <- iv_pairs(c(1, 5), c(3, 7), c(10, 12))

locations <- iv_locate_between(needles, haystack)
iv_align(needles, haystack, locations = locations)
```

---

iv_diff *Diff a vector to create an interval vector*

Description

iv_diff() is a convenient way to generate an iv from a preexisting vector, as long as that vector is in strictly increasing order. It returns an iv that is 1 element shorter than x (unless x is already empty).

It is particularly useful for creating an iv column from an existing column inside of dplyr::mutate(), but requires you to explicitly handle padding in that case, see the examples.

Missing values are allowed, and will be propagated to each side of the resulting interval after applying the diff.

Usage

```
iv_diff(x)
```

Arguments

x [vector]

A vector in strictly increasing order.

Details

`iv_diff()` is inspired by `diff()`.

Value

An iv using `x` as the inner type, with size equal to `max(0L, vec_size(x) - 1L)`.

Examples

```
x <- as.Date("2019-01-01") + c(0, 5, 7, 10, 19)
x

# Notice how the boundaries don't overlap, because the closing `)` aligns
# with an opening `[`.
iv_diff(x)

# Like `iv()`, missing values propagate to both boundaries of the interval.
# Before missing value propagation was applied, it looked like this:
# [1, NA), [NA, 2), [2, 3)
x <- c(1, NA, 2, 3)
iv_diff(x)

# Values in `x` must be in strictly increasing order to generate a valid
# interval vector
x <- c(1, 0, 2, 2)
try(iv_diff(x))

x <- c(1, NA, 0)
try(iv_diff(x))

# ------------------------------------------------------------------------------
# Use with `mutate()`

library(dplyr)

# `iv_diff()` is useful for converting a pre-existing column into an interval
# vector, but you'll need to apply padding to ensure that the size of the
# diff-ed result is the same as the number of rows in your data frame. There
# are two main ways to pad, which are explored below.
df <- tibble(x = c(1, 3, 6))

# Pad with a known lower/upper bound
df %>% mutate(iv = iv_diff(c(0, x)))
df %>% mutate(iv = iv_diff(c(x, Inf)))

# Pad with a missing value, which results in a fully missing interval
df %>% mutate(iv = iv_diff(c(NA, x)))
df %>% mutate(iv = iv_diff(c(x, NA)))
```

iv_format *Formatting*

Description

`iv_format()` is an S3 generic intended as a developer tool for making a custom class print nicely when stored in an iv. The default method simply calls `format()`, and in many cases this is enough for most classes. However, if your class automatically adds justification or padding when formatting a single vector, you might need to implement an `iv_format()` method to avoid that padding, since it often looks strange when nested in an interval vector.

Usage

```
iv_format(x)
```

Arguments

`x`	`[vector]` A vector to format. This will be called on the `iv_start()` and `iv_end()` vectors of an iv.

Value

A character vector, likely generated through a call to `format()`.

Examples

```
# Numeric values get padding automatically through `format()`
x <- c(1, 100)
format(x)

# This ends up looking strange in an iv, so an `iv_format()` method for
# numeric values is implemented which turns off that padding
iv_format(x)
```

iv_pairwise_span *Pairwise span*

Description

`iv_pairwise_span()` computes the *pairwise* "span" between the i-th interval of `x` and the i-th interval of `y`. The pairwise span of two intervals is a new interval containing the minimum start and maximum end of the original intervals. It is similar to `iv_pairwise_set_union()`, except it fills across gaps.

Usage

```
iv_pairwise_span(x, y)
```

Arguments

x, y	[iv] A pair of interval vectors. These will be cast to the same type, and recycled against each other.

Value

An iv the same size and type as x and y.

Examples

```
x <- iv_pairs(c(1, 3), c(6, 8))
y <- iv_pairs(c(5, 7), c(2, 3))

# Can't take the set union when there are gaps
try(iv_pairwise_set_union(x, y))

# But you can compute the span of the intervals
iv_pairwise_span(x, y)
```

---

iv_span *Span*

Description

iv_span() computes the span of an iv. The span is a single interval which encompasses the entire range of the iv. It is similar to iv_groups(), if groups were also merged across gaps.

iv_span() is a *summary* function, like min() and max(), so it always returns a size 1 iv, even for empty ivs. The empty argument can be used to control what is returned in the empty case.

Usage

```
iv_span(x, ..., missing = "propagate", empty = "missing")
```

Arguments

x	[iv] An interval vector.
...	These dots are for future extensions and must be empty.
missing	["propagate" / "drop" / "error" / iv(1)] Handling of missing intervals in x. • "propagate" forces iv_span() to return a missing interval if any missing intervals are detected in x.

- "drop" drops missing intervals before computing the span. If this results in an empty vector, then empty will be applied.
- "error" throws an error if any missing intervals are detected.
- If an iv of size 1 is supplied, then this is returned if any missing intervals are detected. It is cast to the type of x before returning.

empty ["missing" / "error" / iv(1)]

Handling of empty x vectors.

- "missing" forces iv_span() to return a missing interval if x is empty.
- "error" throws an error if x is empty.
- If an iv of size 1 is supplied, then this is returned if x is empty. It is cast to the type of x before returning.

Details

iv_span() is currently limited by the fact that it calls min() and max() internally, which doesn't work for all vector types that ivs supports (mainly data frames). In the future, we hope to be able to leverage vctrs::vec_min() and vctrs::vec_max(), which don't exist yet.

Examples

```
x <- iv_pairs(c(1, 5), c(2, 6), c(9, 10))

# The span covers the full range of values seen in `x`
iv_span(x)

# Compare against `iv_groups()`, which merges overlaps but doesn't merge
# across gaps
iv_groups(x)

x <- iv_pairs(c(1, 3), c(NA, NA), c(5, 6), c(NA, NA))

# Because `iv_span()` is a summary function, if any missing intervals are
# present then it returns a missing interval by default
iv_span(x)

# Further control this with `missing`
iv_span(x, missing = "drop")
try(iv_span(x, missing = "error"))
iv_span(x, missing = iv(-1, 0))

x <- iv(double(), double())

# If `x` is empty, then by default a missing interval is returned
iv_span(x)

# Control this with `empty`
try(iv_span(x, empty = "error"))
iv_span(x, empty = iv(-Inf, Inf))

# `empty` kicks in if `missing = "drop"` is used and all elements were
```

```
# missing
x <- iv(c(NA, NA), c(NA, NA), ptype = double())
iv_span(x, missing = "drop", empty = iv(-Inf, Inf))
```

---

new_iv *Construct a new iv*

Description

new_iv() is a developer focused function for creating a new interval vector. It does minimal checks on the inputs, for performance.

Usage

```
new_iv(start, end, ..., class = character())
```

Arguments

start, end	[vector]
	A pair of vectors to represent the bounds of the intervals.
	To be a valid interval vector, start must be strictly less than end, or both start and end must be a missing value.
...	[name-value pairs]
	Additional named attributes to attach to the result.
class	[character]
	The name of the subclass to create.

Value

A new iv object.

Examples

```
new_iv(1, 2)
```

relation-count *Count relationships between two ivs*

Description

This family of functions counts different types of relationships between two ivs. It works similar to base::match(), where needles[i] checks for a relationship in all of haystack.

- iv_count_overlaps() counts instances of a specific type of overlap between the two ivs.
- iv_count_precedes() counts instances when needles[i] precedes (i.e. comes before) any interval in haystack.
- iv_count_follows() counts instances when needles[i] follows (i.e. comes after) any interval in haystack.

These functions return an integer vector the same size as needles containing a count of the times a particular relationship between the i-th interval of needles and any interval of haystack occurred.

Usage

```
iv_count_overlaps(
  needles,
  haystack,
  ...,
  type = "any",
  missing = "equals",
  no_match = 0L
)

iv_count_precedes(
  needles,
  haystack,
  ...,
  closest = FALSE,
  missing = "equals",
  no_match = 0L
)

iv_count_follows(
  needles,
  haystack,
  ...,
  closest = FALSE,
  missing = "equals",
  no_match = 0L
)
```

Arguments

needles, haystack

[iv]

Interval vectors used for relation matching.

- Each element of needles represents the interval to search for.
- haystack represents the intervals to search in.

Prior to comparison, needles and haystack are coerced to the same type.

...

These dots are for future extensions and must be empty.

type

[character(1)]

The type of relationship to find. One of:

- "any": Finds any overlap whatsoever between an interval in needles and an interval in haystack.
- "within": Finds when an interval in needles is completely within (or equal to) an interval in haystack.
- "contains": Finds when an interval in needles completely contains (or equals) an interval in haystack.
- "equals": Finds when an interval in needles is exactly equal to an interval in haystack.
- "starts": Finds when the start of an interval in needles matches the start of an interval in haystack.
- "ends": Finds when the end of an interval in needles matches the end of an interval in haystack.

missing

[integer(1) / "equals" / "error"]

Handling of missing intervals in needles.

- "equals" considers missing intervals in needles as exactly equal to missing intervals in haystack when determining if there is a matching relationship between them.
- "error" throws an error if any intervals in needles are missing.
- If a single integer value is provided, this represents the count returned for a missing interval in needles. Use 0L to force missing intervals to never match.

no_match

[integer(1) / "error"]

Handling of needles without a match.

- "error" throws an error if any needles have zero matches.
- If a single integer is provided, this represents the count returned for a needle with zero matches. The default value gives unmatched needles a count of 0L.

closest

[TRUE / FALSE]

Should only the closest relationship be returned?

If TRUE, will only return the closest interval(s) in haystack that the current value of needles either precedes or follows. Note that multiple intervals can still be returned if there are ties, which can be resolved using multiple.

Value

An integer vector the same size as `needles`.

See Also

Locating relationships

Examples

```
library(vctrs)

x <- iv_pairs(
  as.Date(c("2019-01-05", "2019-01-10")),
  as.Date(c("2019-01-07", "2019-01-15")),
  as.Date(c("2019-01-20", "2019-01-31"))
)

y <- iv_pairs(
  as.Date(c("2019-01-01", "2019-01-03")),
  as.Date(c("2019-01-04", "2019-01-08")),
  as.Date(c("2019-01-07", "2019-01-09")),
  as.Date(c("2019-01-10", "2019-01-20")),
  as.Date(c("2019-01-15", "2019-01-20"))
)

x
y

# Count the number of times `x` overlaps `y` at all
iv_count_overlaps(x, y)

# Count the number of times `y` is within an interval in `x`
iv_count_overlaps(y, x, type = "within")

# Count the number of times `x` precedes `y`
iv_count_precedes(x, y)

# ------------------------------------------------------------------------

a <- iv(c(1, NA), c(2, NA))
b <- iv(c(NA, NA), c(NA, NA))

# Missing intervals are seen as exactly equal by default, so they are
# considered to overlap
iv_count_overlaps(a, b)

# If you'd like missing intervals to be treated as unmatched, set
# `missing = 0L`
iv_count_overlaps(a, b, missing = 0L)

# If you'd like to propagate missing intervals, set `missing = NA`
iv_count_overlaps(a, b, missing = NA)
```

relation-detect *Detect a relationship between two ivs*

Description

This family of functions detects different types of relationships between two ivs. It works similar to base::%in%, where `needles[i]` checks for a relationship in all of `haystack`.

- `iv_overlaps()` detects a specific `type` of overlap between the two ivs.
- `iv_precedes()` detects if `needles[i]` precedes (i.e. comes before) any interval in `haystack`.
- `iv_follows()` detects if `needles[i]` follows (i.e. comes after) any interval in `haystack`.

These functions return a logical vector the same size as `needles` containing `TRUE` if the interval in `needles` has a matching relationship in `haystack` and `FALSE` otherwise.

Usage

```
iv_overlaps(needles, haystack, ..., type = "any", missing = "equals")

iv_precedes(needles, haystack, ..., missing = "equals")

iv_follows(needles, haystack, ..., missing = "equals")
```

Arguments

`needles`, `haystack`

`[iv]`

Interval vectors used for relation matching.

- Each element of `needles` represents the interval to search for.
- `haystack` represents the intervals to search in.

Prior to comparison, `needles` and `haystack` are coerced to the same type.

`...` These dots are for future extensions and must be empty.

`type` `[character(1)]`

The type of relationship to find. One of:

- `"any"`: Finds any overlap whatsoever between an interval in `needles` and an interval in `haystack`.
- `"within"`: Finds when an interval in `needles` is completely within (or equal to) an interval in `haystack`.
- `"contains"`: Finds when an interval in `needles` completely contains (or equals) an interval in `haystack`.
- `"equals"`: Finds when an interval in `needles` is exactly equal to an interval in `haystack`.
- `"starts"`: Finds when the start of an interval in `needles` matches the start of an interval in `haystack`.

- `"ends"`: Finds when the end of an interval in `needles` matches the end of an interval in `haystack`.

`missing` `[logical(1) / "equals" / "error"]`

Handling of missing intervals in `needles`.

- `"equals"` considers missing intervals in `needles` as exactly equal to missing intervals in `haystack` when determining if there is a matching relationship between them. Matched missing intervals in `needles` result in a `TRUE` value in the result, and unmatched missing intervals result in a `FALSE` value.
- `"error"` throws an error if any intervals in `needles` are missing.
- If a single logical value is provided, this represents the value returned in the result for intervals in `needles` that are missing. You can force missing intervals to be unmatched by setting this to `FALSE`, and you can force them to be propagated by setting this to `NA`.

Value

A logical vector the same size as `needles`.

See Also

Locating relationships

Detecting relationships pairwise

Locating relations from Allen's Interval Algebra

Examples

```
library(vctrs)

x <- iv_pairs(
  as.Date(c("2019-01-05", "2019-01-10")),
  as.Date(c("2019-01-07", "2019-01-15")),
  as.Date(c("2019-01-20", "2019-01-31"))
)

y <- iv_pairs(
  as.Date(c("2019-01-01", "2019-01-03")),
  as.Date(c("2019-01-04", "2019-01-08")),
  as.Date(c("2019-01-07", "2019-01-09")),
  as.Date(c("2019-01-10", "2019-01-20")),
  as.Date(c("2019-01-15", "2019-01-20"))
)

x
y

# Does each interval of `x` overlap `y` at all?
iv_overlaps(x, y)

# Which intervals of `y` are within an interval in `x`?
iv_overlaps(y, x, type = "within")
```

```
# ------------------------------------------------------------------------

a <- iv(c(1, NA), c(2, NA))
b <- iv(c(NA, NA), c(NA, NA))

# Missing intervals are seen as exactly equal by default, so they are
# considered to overlap
iv_overlaps(a, b)

# If you'd like missing intervals to be treated as unmatched, set
# `missing = FALSE`
iv_overlaps(a, b, missing = FALSE)

# If you'd like to propagate missing intervals, set `missing = NA`
iv_overlaps(a, b, missing = NA)
```

`relation-detect-pairwise`

Pairwise detect a relationship between two ivs

Description

This family of functions detects different types of relationships between two ivs *pairwise*, where pairwise means that the i-th interval of `x` is compared against the i-th interval of `y`. This is in contrast to `iv_overlaps()`, which works more like base::%in%.

- `iv_pairwise_overlaps()` detects a specific `type` of overlap between the i-th interval of `x` and the i-th interval of `y`.
- `iv_pairwise_precedes()` detects if the i-th interval of `x` precedes (i.e. comes before) the i-th interval of `y`.
- `iv_pairwise_follows()` detects if the i-th interval of `x` follows (i.e. comes after) the i-th interval of `y`.

These functions return a logical vector the same size as the common size of `x` and `y`.

Usage

```
iv_pairwise_overlaps(x, y, ..., type = "any")

iv_pairwise_precedes(x, y)

iv_pairwise_follows(x, y)
```

Arguments

x, y	[iv] A pair of interval vectors. These will be recycled against each other and cast to the same type.
...	These dots are for future extensions and must be empty.
type	[character(1)] The type of relationship to find. One of: • "any": Finds any overlap whatsoever between an interval in needles and an interval in haystack. • "within": Finds when an interval in needles is completely within (or equal to) an interval in haystack. • "contains": Finds when an interval in needles completely contains (or equals) an interval in haystack. • "equals": Finds when an interval in needles is exactly equal to an interval in haystack. • "starts": Finds when the start of an interval in needles matches the start of an interval in haystack. • "ends": Finds when the end of an interval in needles matches the end of an interval in haystack.

Value

A logical vector the same size as the common size of x and y.

See Also

Locating relationships

Detecting relationships

Locating relations from Allen's Interval Algebra

Examples

```
library(vctrs)

x <- iv_pairs(
  as.Date(c("2019-01-05", "2019-01-10")),
  as.Date(c("2019-01-07", "2019-01-15")),
  as.Date(c("2019-01-20", "2019-01-31"))
)

y <- iv_pairs(
  as.Date(c("2019-01-01", "2019-01-03")),
  as.Date(c("2019-01-07", "2019-01-09")),
  as.Date(c("2019-01-18", "2019-01-21"))
)

x
```

```
y

# Does the i-th interval of `x` overlap the i-th interval of `y`?
iv_pairwise_overlaps(x, y)

# Does the i-th interval of `x` contain the i-th interval of `y`?
iv_pairwise_overlaps(x, y, type = "contains")

# Does the i-th interval of `x` follow the i-th interval of `y`?
iv_pairwise_follows(x, y)

a <- iv_pairs(c(1, 2), c(NA, NA), c(NA, NA))
b <- iv_pairs(c(NA, NA), c(3, 4), c(NA, NA))

# Missing intervals always propagate
iv_pairwise_overlaps(a, b)
```

`relation-locate` *Locate relationships between two ivs*

Description

This family of functions locates different types of relationships between two ivs. It works similar to `base::match()`, where `needles[i]` checks for a relationship in all of `haystack`. Unlike `match()`, *all* matching relationships are returned, rather than just the first.

- `iv_locate_overlaps()` locates a specific `type` of overlap between the two ivs.
- `iv_locate_precedes()` locates where `needles[i]` precedes (i.e. comes before) any interval in `haystack`.
- `iv_locate_follows()` locates where `needles[i]` follows (i.e. comes after) any interval in `haystack`.

These functions return a two column data frame. The `needles` column is an integer vector pointing to locations in `needles`. The `haystack` column is an integer vector pointing to locations in `haystack` with a matching relationship.

Usage

```
iv_locate_overlaps(
  needles,
  haystack,
  ...,
  type = "any",
  missing = "equals",
  no_match = NA_integer_,
  remaining = "drop",
  multiple = "all",
  relationship = "none"
```

```
)

iv_locate_precedes(
  needles,
  haystack,
  ...,
  closest = FALSE,
  missing = "equals",
  no_match = NA_integer_,
  remaining = "drop",
  multiple = "all",
  relationship = "none"
)

iv_locate_follows(
  needles,
  haystack,
  ...,
  closest = FALSE,
  missing = "equals",
  no_match = NA_integer_,
  remaining = "drop",
  multiple = "all",
  relationship = "none"
)
```

Arguments

needles, haystack
: [iv]

  Interval vectors used for relation matching.

  - Each element of needles represents the interval to search for.
  - haystack represents the intervals to search in.

  Prior to comparison, needles and haystack are coerced to the same type.

...
: These dots are for future extensions and must be empty.

type
: [character(1)]

  The type of relationship to find. One of:

  - "any": Finds any overlap whatsoever between an interval in needles and an interval in haystack.
  - "within": Finds when an interval in needles is completely within (or equal to) an interval in haystack.
  - "contains": Finds when an interval in needles completely contains (or equals) an interval in haystack.
  - "equals": Finds when an interval in needles is exactly equal to an interval in haystack.
  - "starts": Finds when the start of an interval in needles matches the start of an interval in haystack.

- "ends": Finds when the end of an interval in needles matches the end of an interval in haystack.

missing [integer(1) / "equals" / "drop" / "error"]

Handling of missing intervals in needles.

- "equals" considers missing intervals in needles as exactly equal to missing intervals in haystack when determining if there is a matching relationship between them.
- "drop" drops missing intervals in needles from the result.
- "error" throws an error if any intervals in needles are missing.
- If a single integer is provided, this represents the value returned in the haystack column for intervals in needles that are missing.

no_match Handling of needles without a match.

- "drop" drops needles with zero matches from the result.
- "error" throws an error if any needles have zero matches.
- If a single integer is provided, this represents the value returned in the haystack column for values of needles that have zero matches. The default represents an unmatched needle with NA.

remaining Handling of haystack values that needles never matched.

- "drop" drops remaining haystack values from the result. Typically, this is the desired behavior if you only care when needles has a match.
- "error" throws an error if there are any remaining haystack values.
- If a single integer is provided (often NA), this represents the value returned in the needles column for the remaining haystack values that needles never matched. Remaining haystack values are always returned at the end of the result.

multiple Handling of needles with multiple matches. For each needle:

- "all" returns all matches detected in haystack.
- "any" returns any match detected in haystack with no guarantees on which match will be returned. It is often faster than "first" and "last" if you just need to detect if there is at least one match.
- "first" returns the first match detected in haystack.
- "last" returns the last match detected in haystack.

relationship Handling of the expected relationship between needles and haystack. If the expectations chosen from the list below are invalidated, an error is thrown.

- "none" doesn't perform any relationship checks.
- "one-to-one" expects:
  - Each value in needles matches at most 1 value in haystack.
  - Each value in haystack matches at most 1 value in needles.
- "one-to-many" expects:
  - Each value in needles matches any number of values in haystack.
  - Each value in haystack matches at most 1 value in needles.
- "many-to-one" expects:
  - Each value in needles matches at most 1 value in haystack.

  - Each value in `haystack` matches any number of values in `needles`.
- `"many-to-many"` expects:
  - Each value in `needles` matches any number of values in `haystack`.
  - Each value in `haystack` matches any number of values in `needles`.

  This performs no checks, and is identical to `"none"`, but is provided to allow you to be explicit about this relationship if you know it exists.
- `"warn-many-to-many"` doesn't assume there is any known relationship, but will warn if `needles` and `haystack` have a many-to-many relationship (which is typically unexpected), encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying `"many-to-many"`.

`relationship` is applied after `filter` and `multiple` to allow potential multiple matches to be filtered out first.

`relationship` doesn't handle cases where there are zero matches. For that, see `no_match` and `remaining`.

`closest` `[TRUE / FALSE]`

Should only the closest relationship be returned?

If `TRUE`, will only return the closest interval(s) in `haystack` that the current value of `needles` either precedes or follows. Note that multiple intervals can still be returned if there are ties, which can be resolved using `multiple`.

Value

A data frame containing two integer columns named `needles` and `haystack`.

See Also

Detecting relationships

Detecting relationships pairwise

Locating relations from Allen's Interval Algebra

Examples

```
x <- iv_pairs(
  as.Date(c("2019-01-05", "2019-01-10")),
  as.Date(c("2019-01-07", "2019-01-15")),
  as.Date(c("2019-01-20", "2019-01-31"))
)

y <- iv_pairs(
  as.Date(c("2019-01-01", "2019-01-03")),
  as.Date(c("2019-01-04", "2019-01-08")),
  as.Date(c("2019-01-07", "2019-01-09")),
  as.Date(c("2019-01-10", "2019-01-20")),
  as.Date(c("2019-01-15", "2019-01-20"))
)

x
y
```

```
# Find any overlap between `x` and `y`
loc <- iv_locate_overlaps(x, y)
loc

iv_align(x, y, locations = loc)

# Find where `x` contains `y` and drop results when there isn't a match
loc <- iv_locate_overlaps(x, y, type = "contains", no_match = "drop")
loc

iv_align(x, y, locations = loc)

# Find where `x` precedes `y`
loc <- iv_locate_precedes(x, y)
loc

iv_align(x, y, locations = loc)

# Filter down to find only the closest interval in `y` of all the intervals
# where `x` preceded it
loc <- iv_locate_precedes(x, y, closest = TRUE)

iv_align(x, y, locations = loc)

# Note that `closest` can result in duplicates if there is a tie.
# `2019-01-20` appears as an end date twice in `haystack`.
loc <- iv_locate_follows(x, y, closest = TRUE)
loc

iv_align(x, y, locations = loc)

# Force just one of the ties to be returned by using `multiple`.
# Here we just request any of the ties, with no guarantee on which one.
loc <- iv_locate_follows(x, y, closest = TRUE, multiple = "any")
loc

iv_align(x, y, locations = loc)

# ------------------------------------------------------------------------------

a <- iv(NA, NA)
b <- iv(c(NA, NA), c(NA, NA))

# By default, missing intervals in `needles` are seen as exactly equal to
# missing intervals in `haystack`, which means that they overlap
iv_locate_overlaps(a, b)

# If you'd like missing intervals in `needles` to always be considered
# unmatched, set `missing = NA`
iv_locate_overlaps(a, b, missing = NA)
```

vector-count *Count relationships between a vector and an iv*

Description

This family of functions counts different types of relationships between a vector and an iv. It works similar to base::match(), where needles[i] checks for a match in all of haystack.

- iv_count_between() counts instances of when needles, a vector, falls between the bounds of haystack, an iv.
- iv_count_includes() counts instances of when needles, an iv, includes the values of haystack, a vector.

These functions return an integer vector the same size as needles containing a count of the times where the i-th value of needles contained a match in haystack.

Usage

```
iv_count_between(needles, haystack, ..., missing = "equals", no_match = 0L)

iv_count_includes(needles, haystack, ..., missing = "equals", no_match = 0L)
```

Arguments

needles, haystack
: [vector, iv]

  For iv_*_between(), needles should be a vector and haystack should be an iv.

  For iv_*_includes(), needles should be an iv and haystack should be a vector.

  - Each element of needles represents the value / interval to match.
  - haystack represents the values / intervals to match against.

...
: These dots are for future extensions and must be empty.

missing
: [integer(1) / "equals" / "error"]

  Handling of missing values in needles.

  - "equals" considers missing values in needles as exactly equal to missing values in haystack when determining if there is a matching relationship between them.
  - "error" throws an error if any values in needles are missing.
  - If a single integer value is provided, this represents the count returned for a missing value in needles. Use 0L to force missing values to never match.

no_match
: [integer(1) / "error"]

  Handling of needles without a match.

  - "error" throws an error if any needles have zero matches.
  - If a single integer is provided, this represents the count returned for a needle with zero matches. The default value gives unmatched needles a count of 0L.

Value

An integer vector the same size as `needles`.

See Also

Locating relationships between a vector and an iv

Examples

```
x <- as.Date(c("2019-01-05", "2019-01-10", "2019-01-07", "2019-01-20"))

y <- iv_pairs(
  as.Date(c("2019-01-01", "2019-01-03")),
  as.Date(c("2019-01-04", "2019-01-08")),
  as.Date(c("2019-01-07", "2019-01-09")),
  as.Date(c("2019-01-10", "2019-01-20")),
  as.Date(c("2019-01-15", "2019-01-20"))
)

x
y

# Count the number of times `x` is between the intervals in `y`
iv_count_between(x, y)

# Count the number of times `y` includes a value from `x`
iv_count_includes(y, x)

# ------------------------------------------------------------------------------

a <- c(1, NA)
b <- iv(c(NA, NA), c(NA, NA))

# By default, missing values in `needles` are treated as being exactly
# equal to missing values in `haystack`, so the missing value in `a` is
# considered between the missing interval in `b`.
iv_count_between(a, b)
iv_count_includes(b, a)

# If you'd like to propagate missing values, set `missing = NA`
iv_count_between(a, b, missing = NA)
iv_count_includes(b, a, missing = NA)

# If you'd like missing values to be treated as unmatched, set
# `missing = 0L`
iv_count_between(a, b, missing = 0L)
iv_count_includes(b, a, missing = 0L)
```

vector-detect *Detect relationships between a vector and an iv*

Description

This family of functions detects different types of relationships between a vector and an iv. It works similar to base::%in%, where needles[i] checks for a match in all of haystack.

- iv_between() detects when needles, a vector, falls between the bounds in haystack, an iv.
- iv_includes() detects when needles, an iv, includes the values of haystack, a vector.

This function returns a logical vector the same size as needles containing TRUE if the value in needles matches any value in haystack and FALSE otherwise.

Usage

```
iv_between(needles, haystack, ..., missing = "equals")

iv_includes(needles, haystack, ..., missing = "equals")
```

Arguments

needles, haystack
: [vector, iv]

  For iv_*_between(), needles should be a vector and haystack should be an iv.

  For iv_*_includes(), needles should be an iv and haystack should be a vector.

  - Each element of needles represents the value / interval to match.
  - haystack represents the values / intervals to match against.

...
: These dots are for future extensions and must be empty.

missing
: [logical(1) / "equals" / "error"]

  Handling of missing values in needles.

  - "equals" considers missing values in needles as exactly equal to missing values in haystack when determining if there is a matching relationship between them. Matched missing values in needles result in a TRUE value in the result, and unmatched missing values result in a FALSE value.
  - "error" throws an error if any values in needles are missing.
  - If a single logical value is provided, this represents the value returned in the result for values in needles that are missing. You can force missing values to be unmatched by setting this to FALSE, and you can force them to be propagated by setting this to NA.

Value

A logical vector the same size as needles.

See Also

Locating relationships

Locating relationships between a vector and an iv

Pairwise detect relationships between a vector and an iv

Examples

```
x <- as.Date(c("2019-01-05", "2019-01-10", "2019-01-07", "2019-01-20"))

y <- iv_pairs(
  as.Date(c("2019-01-01", "2019-01-03")),
  as.Date(c("2019-01-04", "2019-01-08")),
  as.Date(c("2019-01-07", "2019-01-09")),
  as.Date(c("2019-01-10", "2019-01-20")),
  as.Date(c("2019-01-15", "2019-01-20"))
)

x
y

# Detect if the i-th location in `x` is between any intervals in `y`
iv_between(x, y)

# Detect if the i-th location in `y` includes any value in `x`
iv_includes(y, x)

# ------------------------------------------------------------------------

a <- c(1, NA)
b <- iv(c(NA, NA), c(NA, NA))

# By default, missing values in `needles` are treated as being exactly
# equal to missing values in `haystack`, so the missing value in `a` is
# considered between the missing interval in `b`.
iv_between(a, b)
iv_includes(b, a)

# If you'd like to propagate missing values, set `missing = NA`
iv_between(a, b, missing = NA)
iv_includes(b, a, missing = NA)

# If you'd like missing values to be treated as unmatched, set
# `missing = FALSE`
iv_between(a, b, missing = FALSE)
iv_includes(b, a, missing = FALSE)
```

`vector-detect-pairwise`

Pairwise detect relationships between a vector and an iv

Description

This family of functions detects different types of relationships between a vector and an iv *pairwise*. where pairwise means that the i-th value of `x` is compared against the i-th value of `y`. This is in contrast to `iv_between()`, which works more like base::%in%.

- `iv_pairwise_between()` detects if the i-th value of `x`, a vector, falls between the bounds of the i-th value of `y`, an iv.
- `iv_pairwise_includes()` detects if the i-th value of `x`, an iv, includes the i-th value of `y`, a vector.

These functions return a logical vector the same size as the common size of `x` and `y`.

Usage

```
iv_pairwise_between(x, y)

iv_pairwise_includes(x, y)
```

Arguments

`x`, `y`	`[vector, iv]` For `iv_pairwise_between()`, `x` must be a vector and `y` must be an iv. For `iv_pairwise_includes()`, `x` must be an iv and `y` must be a vector. `x` and `y` will be recycled against each other.

Value

A logical vector the same size as the common size of `x` and `y`.

See Also

Locating relationships

Locating relationships between a vector and an iv

Detecting relationships between a vector and an iv

Examples

```
x <- as.Date(c("2019-01-01", "2019-01-08", "2019-01-21"))

y <- iv_pairs(
  as.Date(c("2019-01-01", "2019-01-03")),
  as.Date(c("2019-01-07", "2019-01-09")),
  as.Date(c("2019-01-18", "2019-01-21"))
)

x
y

# Does the i-th value of `x` fall between the i-th interval of `y`?
iv_pairwise_between(x, y)
```

```
# Does the i-th interval of `y` include the i-th value of `x`?
iv_pairwise_includes(y, x)

a <- c(1, NA, NA)
b <- iv_pairs(c(NA, NA), c(3, 4), c(NA, NA))

# Missing intervals always propagate
iv_pairwise_between(a, b)
iv_pairwise_includes(b, a)
```

`vector-locate` *Locate relationships between a vector and an iv*

Description

This family of functions locates different types of relationships between a vector and an iv. It works similar to `base::match()`, where `needles[i]` checks for a match in all of `haystack`. Unlike `match()`, *all* matches are returned, rather than just the first.

- `iv_locate_between()` locates where `needles`, a vector, falls between the bounds of `haystack`, an iv.
- `iv_locate_includes()` locates where `needles`, an iv, includes the values of `haystack`, a vector.

These functions return a two column data frame. The `needles` column is an integer vector pointing to locations in `needles`. The `haystack` column is an integer vector pointing to locations in `haystack` with a match.

Usage

```
iv_locate_between(
  needles,
  haystack,
  ...,
  missing = "equals",
  no_match = NA_integer_,
  remaining = "drop",
  multiple = "all",
  relationship = "none"
)

iv_locate_includes(
  needles,
  haystack,
  ...,
  missing = "equals",
  no_match = NA_integer_,
```

```
  remaining = "drop",
  multiple = "all",
  relationship = "none"
)
```

Arguments

needles, haystack

[vector, iv]

For iv_*_between(), needles should be a vector and haystack should be an iv.

For iv_*_includes(), needles should be an iv and haystack should be a vector.

- Each element of needles represents the value / interval to match.
- haystack represents the values / intervals to match against.

...

These dots are for future extensions and must be empty.

missing

[integer(1) / "equals" / "drop" / "error"]

Handling of missing values in needles.

- "equals" considers missing values in needles as exactly equal to missing values in haystack when determining if there is a matching relationship between them.
- "drop" drops missing values in needles from the result.
- "error" throws an error if any values in needles are missing.
- If a single integer is provided, this represents the value returned in the haystack column for values in needles that are missing.

no_match

Handling of needles without a match.

- "drop" drops needles with zero matches from the result.
- "error" throws an error if any needles have zero matches.
- If a single integer is provided, this represents the value returned in the haystack column for values of needles that have zero matches. The default represents an unmatched needle with NA.

remaining

Handling of haystack values that needles never matched.

- "drop" drops remaining haystack values from the result. Typically, this is the desired behavior if you only care when needles has a match.
- "error" throws an error if there are any remaining haystack values.
- If a single integer is provided (often NA), this represents the value returned in the needles column for the remaining haystack values that needles never matched. Remaining haystack values are always returned at the end of the result.

multiple

Handling of needles with multiple matches. For each needle:

- "all" returns all matches detected in haystack.
- "any" returns any match detected in haystack with no guarantees on which match will be returned. It is often faster than "first" and "last" if you just need to detect if there is at least one match.

- `"first"` returns the first match detected in `haystack`.
- `"last"` returns the last match detected in `haystack`.

`relationship` Handling of the expected relationship between `needles` and `haystack`. If the expectations chosen from the list below are invalidated, an error is thrown.

- `"none"` doesn't perform any relationship checks.
- `"one-to-one"` expects:
  - Each value in `needles` matches at most 1 value in `haystack`.
  - Each value in `haystack` matches at most 1 value in `needles`.
- `"one-to-many"` expects:
  - Each value in `needles` matches any number of values in `haystack`.
  - Each value in `haystack` matches at most 1 value in `needles`.
- `"many-to-one"` expects:
  - Each value in `needles` matches at most 1 value in `haystack`.
  - Each value in `haystack` matches any number of values in `needles`.
- `"many-to-many"` expects:
  - Each value in `needles` matches any number of values in `haystack`.
  - Each value in `haystack` matches any number of values in `needles`.

  This performs no checks, and is identical to `"none"`, but is provided to allow you to be explicit about this relationship if you know it exists.
- `"warn-many-to-many"` doesn't assume there is any known relationship, but will warn if `needles` and `haystack` have a many-to-many relationship (which is typically unexpected), encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying `"many-to-many"`.

`relationship` is applied after `filter` and `multiple` to allow potential multiple matches to be filtered out first.

`relationship` doesn't handle cases where there are zero matches. For that, see `no_match` and `remaining`.

Value

A data frame containing two integer columns named `needles` and `haystack`.

See Also

Locating relationships

Detect relationships between a vector and an iv

Pairwise detect relationships between a vector and an iv

Examples

```
x <- as.Date(c("2019-01-05", "2019-01-10", "2019-01-07", "2019-01-20"))

y <- iv_pairs(
  as.Date(c("2019-01-01", "2019-01-03")),
  as.Date(c("2019-01-04", "2019-01-08")),
  as.Date(c("2019-01-07", "2019-01-09")),
```

```
  as.Date(c("2019-01-10", "2019-01-20")),
  as.Date(c("2019-01-15", "2019-01-20"))
)

x
y

# Find any location where `x` is between the intervals in `y`
loc <- iv_locate_between(x, y)
loc

iv_align(x, y, locations = loc)

# Find any location where `y` includes the values in `x`
loc <- iv_locate_includes(y, x)
loc

iv_align(y, x, locations = loc)

# Drop values in `x` without a match
loc <- iv_locate_between(x, y, no_match = "drop")
loc

iv_align(x, y, locations = loc)

# ------------------------------------------------------------------------

a <- c(1, NA)
b <- iv(c(NA, NA), c(NA, NA))

# By default, missing values in `needles` are treated as being exactly
# equal to missing values in `haystack`, so the missing value in `a` is
# considered between the missing interval in `b`.
iv_locate_between(a, b)
iv_locate_includes(b, a)

# If you'd like missing values in `needles` to always be considered
# unmatched, set `missing = NA`
iv_locate_between(a, b, missing = NA)
iv_locate_includes(b, a, missing = NA)
```

Index