Wire-Cell Toolkit Rectangles

Brett Viren

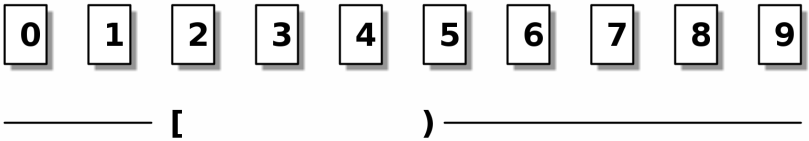
September 12, 2022

Topics

- Interval trees, sets and maps
- Boost Implementation
- Extend to 2D: Rectangles
- Application to deghosting

Intervals

Consider the 1D domain, either $x \in \mathbb{R}$ or $i \in \mathbb{I}$.



An *interval* $[x_1, x_2)$ is some *right-open*, finite, contiguous subset $x_1 \leq x < x_2$.

- Example, $[2, 5)$ 2 is in the interval, **5 is not**,
  - *ie* like Python/C++ iterators, "end" is just-past last,
  - $[2, 5)$ and $[5, 7)$ are not overlapping.
- Also possible, but ignored here: *closed* $[a, b]$, *open* (a, b) or *left-open* $(a, b]$.

Interval tree - a binary tree of intervals

Interval tree provides optimized storage and query of intervals.

- For n stored intervals, naive operations require at least $\mathcal{O}(n)$.

With a tree structure and queries returning m intervals, expect:

insert/delete	$\mathcal{O}(\log n)$
full creation	$\mathcal{O}(n \log n)$
point query	$\mathcal{O}(\log n + m)$
interval query	$\mathcal{O}(\log n)$
memory	$\mathcal{O}(n)$

For brief description of the data structure and algorithms,
https://en.wikipedia.org/wiki/Interval_tree

Interval set

Adds *set-theoretic* operations: *union* (addition), *difference* (subtraction, symmetric or asymmetric) and *intersection*.

Set-addition, result is governed by an *interval combining style*. Given:

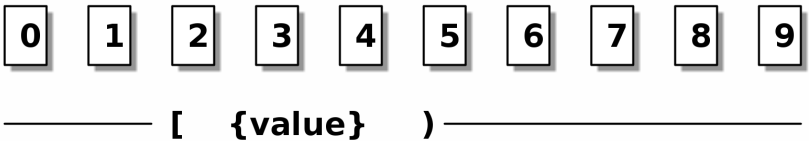
$$s_1 \equiv \{[1, 4)\},\ s_2 \equiv \{[2, 5)\},\ s_3 \equiv \{[5, 7)\}$$

Then $s_1 + s_2 + s_3$ depends on one of three styles:

joining	$\{[1, 7)\}$ (combine on overlap and touch)
separating	$\{[1, 5), [5, 7)\}$ (combine on overlap, not touch)
splitting	$\{[1, 2), [2, 4), [4, 5), [5, 7)\}$ (split on overlap, keep borders)

Note: interval sets have no value associated to intervals. . . .

Interval maps - associate values with intervals.



An interval map still holds intervals as interval sets.
And the interval's value(s) is itself held in a **set**.

Interval map: aggregation on overlap

Differing values aggregate same regardless of *interval combining style*:

$$m_1 \equiv \{[1,4) \to 0\},\ m_2 \equiv \{[2,5) \to 1\},\ m_3 \equiv \{[5,7) \to 2\}$$

Then $m_1 + m_2 + m_3$ produces the following independent of the style:

$$([1,2) \to \{0\}), ([2,4) \to \{0,1\}), ([4,5) \to \{1\}), ([5,7) \to \{2\})\}$$

Note, overlap $[2,4)$ has set with two values $\{0,1\}$.

Common values are subject to *interval combining style*:

$$m_1 \equiv \{[1,4) \to 0\},\ m_2 \equiv \{[2,5) \to 0\},\ m_3 \equiv \{[5,7) \to 0\}$$

Then $m_1 + m_2 + m_3$ depends on one of just two styles:

joining $\{([1,7) \to 0)\}$

splitting $\{([1,2) \to 0), ([2,4) \to \{0\}), ([4,5) \to \{0\}), ([5,7) \to \{0\})\}$

Boost Interval Container Library (`boost::icl`)

- https://www.boost.org/doc/libs/master/libs/icl
- Supports containers: intervals, interval sets and interval maps.
- Supports the *interval combining styles* described above.
- Provides family of free functions and operators.
  - creation, set operations, queries, iteration
  - All functions work on all types of containers, where it makes sense.

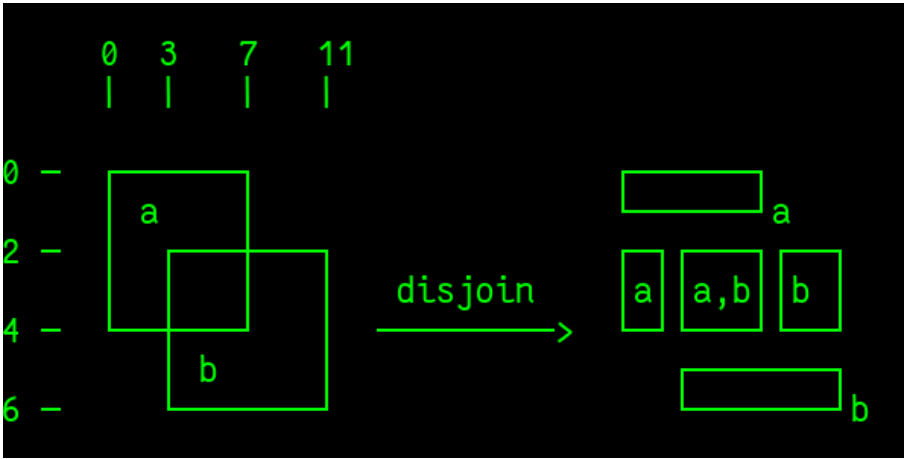
boost::icl interval map example

```
using key_t = int;    // the interval domain type
using val_t = double; // associated value type
using set_t = std::set<val_t>;
// with "joining" combining style:
using imap_t = boost::icl::interval_map<key_t, set_t>;
// or with "splitting" combinging style:
// using imap_t = boost::icl::split_interval_map<key_t, set_t>;
using interval_t = imap_t::interval_type;

imap_t m;
m += std::make_pair(interval_t::right_open(0, 7), 42.0);

auto qi = interval_t::right_open(1, 2);
auto mq = m & qi; // "bitwise and"
for (const auto& [i, s] : mq) {
    cout << "in interval " << i << " we have set {";
    for (const auto& v : s) { cout << " " << v; }
    cout << " }\n";
}
```

Extend interval map to 2D – `WireCell::Rectangles`



(thank `herbstluftwm` for the artwork)

A dimensional hierarchy implementation of 2D interval maps

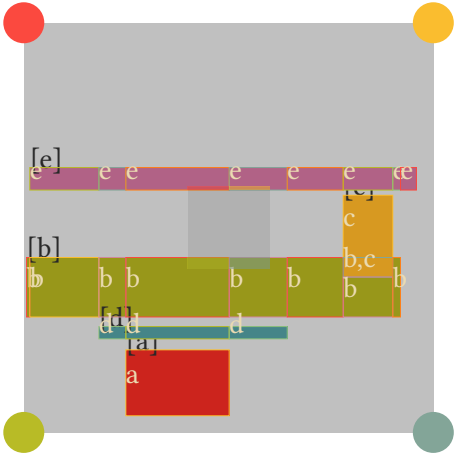
Very simple implementation.

```
using xkey_t = ...;
using ykey_t = ...;
using value_t = ...;
using set_t = std::set<value_t>;
// Intervals along the vertical Y-axis
using ymap_t = boost::icl::interval_map<ykey_t, set_t>;
// Intervals along the horizontal X-axis
using xmap_t = boost::icl::interval_map<xkey_t, ymap_t>;
```

Essentially, an X-dimension map to a Y-dimension map to a set of values.
That's really all there is.

- `WireCell::Rectangles` provides a light dusting of syntactic sugar.

Example - random rectangles



- Initial rectangles: solid color holding black letter data, *eg*: [a].
- 2D overlaps are out outlined rectangles holding a set of white letters.
- Central gray square is a query:

```
X<- [400,600) Y<- [400,600)
X-> [400,501.495) Y-> [400,408.36) {e}
X-> [400,501.495) Y-> [572.966,600) {b}
X-> [501.495,600) Y-> [400,408.36) {e}
X-> [501.495,600) Y-> [572.966,600) {b}
```

The *dimensional hierarchy* implementation results in each X-interval projecting across **all** rectangles. Rectangles which may not overlap in the Y-dimension nonetheless become segmented along the X-dimension. This increases the complexity-n. Will it be prohibitive?

Cluster Shadows - apply `Rectangles`

A **Geometric Cluster** is a *connected component* of a `b-b` graph where:

- `b-b` edges formed between `b` nodes in neighboring slices which overlap.
- `ICluster` already has this `b-b` subgraph embedded.
- TODO: form these `b-b` edges given dead/bad channels/blobs.

A **Blob Shadow** (see last presentation).

- Describes overlap of two blobs in one view.
- Can be wire-type or channel-type shadow.
- Results in a (different) `b-b` **blob shadow graph**, edge is the shadow.

A **Cluster Shadow Graph** combines these two `b-b` graphs.

- Makes a `g-g` graph, each `g` is "geometric cluster".
- A `g` holds one `Rectangles`, each giving the cluster's projection in a view.
- A `g-g` edge for any cluster pair with non-zero shadow.

Application: blob deghosting

A ghost blob truly has no charge and tends to have shadows with real blobs.
Use the `Rectangles` from Cluster Shadows (CS) to find them:

- Iterate over `g-g` edges to check pair-wise, per-view cluster shadows.
- Use set-difference and/or set-intersection of the CS `Rectangles`.
- Possible metrics: area or charge*area or charge/area in diff/inter to total.
- Define selection criteria for ghosts.
  - compare against `BlobDepoFill` true charge blobs

This work is still TODO.

- Functions to find frame charge in 2D intervals of `Rectangles`.
- Develop and choose the metrics and their selection criteria