Reference Manual for Generic Data Structure for the Rectangle Intersection Problem

Vikas Kapoor *   Alexander Wolff *

Revision: 1.1, Date: 1998/01/22

*Institut für Informatik, Fachbereich Mathematik und Informatik, Freie Universität Berlin, Takustraße 9, 14195 Berlin, Germany, {kapoor | awolff}@inf.fu-berlin.de

Introduction

An algorithm for the rectangle intersection problem can be sketched as follows: initialize the interval tree with the intervals (on the y-axis) to which the left sides of the rectangles correspond. Then sweep horizontally across the plane. When the sweep line hits a rectangle, query with the corresponding interval and insert it into the interval tree. An interval is deleted when the sweep line leaves the corresponding rectangle.

As usual, the sweep line is supported by two data structures, the event-point schedule and the sweep-line status.

Event points are the x-values where the sweep line must stop because either its status changes or intersections have to be reported. In our case all event points are known before the sweep begins: they are the x-values of left and right edges of rectangles. It suffices to use a simple list of event points sorted in ascending x-order to implement our event-point schedule. An event point must be stored in such a way that it is clear whether it refers to a left or right edge of a rectangle.

The sweep-line status stores intervals corresponding to the intersections of the sweep line with rectangles. Initially the sweep-line status is empty. When a left edge of a rectangle is encountered during the sweep it is inserted into the sweep-line status. All rectangles whose edges are present in the sweep-line status and intersect the interval represented by this edge are reported. When a right edge of a rectangle is encountered, the left edge is deleted from the sweep-line status.

This reduces the rectangle intersection problem to the problem of maintaining a set of intervals such that intervals can be efficiently inserted and deleted, and queries asking for all intervals that intersect a given interval can be answered quickly. To achieve this, we implement our sweep-line status with the interval tree data structure. This data structure allows inserting and deleting in $\mathcal{O}(\log n)$ time and reporting all intervals that intersect a given interval in $\mathcal{O}(k + \log n)$ time, where k is the number of intersections. The space requirement for the interval tree is linear.

Chapter 1

Rectangle Intersection

Definition

Class *GenRecInters*<...> reports intersection among input rectangles.

#include <*gen_rec_inters.h*>

Template Parameters

The class *GenRecInters*<...> is templated with the following parameters:

parameter	*task*	*requirement*
ForwardIt	access to input rectangles	STL forward iterator
LLXDA	access to x-coordinate of the lower left cornerpoint of a label candidate	read data accessor
LLYDA	access to y-coordinate of the lower left cornerpoint of a label candidate	read data accessor
URXDA	access to x-coordinate of the upper right cornerpoint of a label candidate	read data accessor
URYDA	access to y-coordinate of the upper right cornerpoint of a label candidate	read data accessor

Table 1.1: template parameters of the class `GenRecInters`

Inherits From

GenRecInters<...> inherits from the base class *GenRecInters_user_compare*<...> which differs from *GenRecInters*<...> only in that it needs two further template parameters to define an intersection between two rectangles, see Chapter 2. This class hierarchy is necessary due to the fact that most compilers cannot handle default template parameters yet.

The constructor of *GenRecInters*<...> calls the constructor of the base class by plugging data accessors into the template class which implements the compare function objects described in Chapters 3 and 4.

Types

GenRecInters<...>:: LeftDA::value_type value_type;

Type of the coordinates of the cornerpoints of the input rectangles.

GenRecInters<...>:: Rectangle_it

Type of the iterator that allows to traverse the rectangle intersecting a given rectangle.

GenRecInters<...>:: enum Algo_state { *none, rectangle_start, rectangle_end, done* }

states of the algorithm.

Creation

GenRecInters<...> rec_inters(ForwardIt beg,
ForwardIt end,
LLXDA lxda = LLXDA(),
LLYDA lyda = LLYDA(),
URXDA hxda = URXDA(),
URYDA hyda = URYDA())

creates the object *rec_inters* of type *GenRecInters<...>*.

Loop-Kernel

bool	*rec_inters.valid()*	is false if the algorithm is finished.
Algo_state	*rec_inters.step()*	pushes *rec_inters* to next event point.

Access Functions

ForwardIt	*rec_inters.current()*	returns an iterator referring to the current rectangle in *rec_inters*.
Rectangle_it	*rec_inters.begin()*	returns an iterator referring to the first rectangle intersecting the current rectangle in *rec_inters*.
Rectangle_it	*rec_inters.end()*	returns an iterator which marks the end of the sequence of rectangles intersecting the current rectangle.
int	*rec_inters.current_num_of_conflicts()*	returns the number of rectangles intersecting the current rectangle.
int	*rec_inters.num_of_conflicts()*	returns the total number of intersections.

Chapter 2

Rectangle Intersection with User-Supplied Intersection Definition

Definition

The class *GenRectInters_user_compare*<...> is similar to the class *GenRecInters*<...> with the only difference that the user is expected to provide two further template parameters which define an intersection between two rectangles.

#include <*gen_rec_inters.h*>

Template Parameters

Apart from the template parameters of Table 1.1, the class *GenRectInters_user_compare*<...> is templated with the following parameters:

parameter	*task*	*requirement*
CompareX	compare function object for the x-coordinates of vertical rectangle edges	function object
CompareY	compare function object for the y-coordinates of horizontal rectangle edges	function object

Table 2.1: template parameters of the class `GenRecInters_user_intersection`

Creation

GenRectInters_user_compare<...> *rec_inters_user_comp*(*ForwardIt beg,*
ForwardIt end,
LeftDA lda = LeftDA(),
RightDA rda = RightDA(),
CompareX com_x = CompareX(),
CompareY com_y = CompareY())

creates the object *rec_inters_user_comp* of type *GenRectInters_user_compare*<...>.

Chapter 3

Comparing the Vertical Edges of Rectangles

Definition

Given two pairs (x_1, l_1) and (x_2, l_2), where x_1 and x_2 are x-coordinates of vertical rectangle edges, and l_1 and l_2 are iterators pointing to the corresponding rectangles. Then a function object of type *Comp_x<ForwardIt, LLXDA>* returns *true* if $x_1 < x_2$ or if $x_1 = x_2$ and x_2 is the x-coordinate of the *left* edge of a rectangle.

#include <gen_rec_inters.h>

Template Parameters

The class *Comp_x<ForwardIt, LLXDA>* is templated with *ForwardIt* and *LLXDA* described in Table 1.1.

Types

Comp_x<ForwardIt, LLXDA>:: LLXDA::value_type value_type;

Type of the x-coordinates of input rectangles.

Comp_x<ForwardIt, LLXDA>:: pair<value_type, ForwardIt > Pair;

Type of a pair to be compared by the function object.

Creation

Comp_x<ForwardIt, LLXDA> comp_x(LLXDA llxda = LLXDA());

creates the object *comp_x* of type *Comp_x<ForwardIt, LLXDA>*.

Operations

bool	*comp_x(Pair p, Pair q)*	Let $p = (x_1, l_1)$ and $q = (x_2, l_2)$, where x_1 and x_2 are x-coordinates of vertical rectangle edges, and l_1 and l_2 are iterators pointing to the corresponding rectangles. The method returns *true* if $x_1 < x_2$ or if $x_1 = x_2$ and x_2 is the x-coordinate of the *left* edge of l_2.

Chapter 4

Comparing the Horizontal Edges of Labels

Definition

Given the y-coordinates y_1 and y_2 of the horizontal edges of two rectangles, a function object of type *Comp_y<DA>* returns *true* iff $y_1 < y_2$.

#include <gen_rec_inters.h>

Template Parameters

The class *Comp_y<DA>* is templated with a data accessor *DA* whose *value_type* must be that of the y-coordinates to be compared.

Types

Comp_y<DA>:: typedef DA::value_type value_type;

Type of the y-coordinates of vertical edges of rectangles.

Creation

Comp_y<DA> *comp_y*;

creates the object *comp_y* of type *Comp_y<DA>*.

Operations

bool *comp_y(value_type y1, value_type y2)*

returns *true* iff *y1* < *y2*.

Chapter 5

Example

The following example illustrates the use of interval tree.

```
#if (__SUNPRO_CC)
#define typename
#endif

#include <iostream.h>
#include <fstream.h>
#include <vector.h>
#include <gen_rec_inters.h>

/* type of a rectangle */
struct Rectangle
{
  double x1, x2, y1, y2;
  Rectangle()
  {}
  Rectangle(double x_l, double x_h, double y_l, double y_h)
  :  x1(x_l), x2(x_h), y1(y_l), y2(y_h)
  {}
};

/* input operator for  rectangle: */
istream &operator>> (istream &in, Rectangle& rec)
{
  in >> rec.x1 >> rec.y1 >> rec.x2 >> rec.y2;
  return in;
}

/* output operator for rectangle: */
ostream &operator<< (ostream &out, Rectangle const &rec)
{
  out << "(" << rec.x1 << "," << rec.y1 << ")==("
      << rec.x2 << "," << rec.y2 << ")";
  return out;
}
```

```
/* data accessors for rectangles */
struct lowXcoord { typedef double value_type; };
struct lowYcoord { typedef double value_type; };
struct highXcoord { typedef double value_type; };
struct highYcoord { typedef double value_type; };

/* the corresponding data accessing functions */
template <class It>
inline double get(lowXcoord da, It const &it) { return (*it).  x1; }
template <class It>
inline double get(lowYcoord da, It const &it) { return (*it).  y1; }
template <class It>
inline double get(highXcoord da, It const &it) { return (*it).  x2; }
template <class It>
inline double get(highYcoord da, It const &it) { return (*it).  y2; }


main(int argc, char *argv[])
{
    /* type definitions */
    typedef typename list<Rectangle>::iterator Rec_it;
    typedef GenRecInters< Rec_it, lowXcoord, lowYcoord, highXcoord, highYcoord>
    Rec_algo;
    typedef typename Rec_algo::Rectangle_it Recalgo_it;

    cout << "This program detects pairs of properly intersecting rectangles."
     << endl << endl;

    /* read the input rectangles */
    list<Rectangle> rectangles;
    cout << "Please enter the coordinates of lower left and upper right"
         << endl << "corner of the rectangles (ll x, ll y, ur x, ur y)."
         << endl << endl;
    Rectangle rectangle;
    while (cin >> rectangle)
        rectangles.push_back(rectangle);

    /* initialize the algorithm */
    cout << endl << "initializing..." << endl;
    Rec_algo my_recalgo(rectangles.begin(), rectangles.end());
    /* the sweep */
    cout << "sweeping..." << endl;
    do
    {
      Rec_algo::Algo_state state;
      switch ( state = my_recalgo.step() )
      {
      case Rec_algo::rectangle_start:
        {
          cout << " rectangle " << *my_recalgo.current()
               << " inserted in sls" << endl;
          Recalgo_it current = my_recalgo.begin();
          for ( ; !(current == my_recalgo.end()); ++current)
            cout << "  intersects rectangle " << **current << endl;
```

```
        }
        break;
      case Rec_algo::rectangle_end:
        cout << " rectangle " << *my_recalgo.current()
             << " deleted from sls" << endl;
        break;
      default:
        cout << " state of Iso_rect_inter = " << state << endl;
      }
    } while (my_recalgo.valid());
    cout << "done." << endl;
}
```