Las Vegas Geometry

Eray Özkural

February 7, 2017

1 Introduction

This project is concerned with implementation of a general purpose randomized scheme for a variety of geometric algorithms. The work draws from recent results regarding properties of random subsets of geometric objects. In the paper I have been studying the authors propose a number of new non-deterministic algorithms, some of which have better bounds than previously known algorithms. The interesting aspect of the algorithms is that they share a general structure, and their expected running times arise from sharp bounds of this general structure.

Among these algorithms is an $O(A+nlogn)$ algorithm to find all intersecting pairs of a set of line segments and another algorithm computes the convex hull of a set of points in $O(nlogn)$ for $d = 3$ and $O(n^{\lfloor d/2 \rfloor})$ for $d > 3$. [1] The diameter of a set of points in E^3 is computed in $O(nlogn)$ time and also included is an algorithm for half-space range reporting. The paper has in addition theoretical results such as tight bounds for $(\leq k)$-sets and proofs for higher order Voronoi diagrams.

As such, the paper in fact condenses a large number of results in a technical exposition and it is impossible to test all the ideas because of its breadth. Many problems are tackled. For instance three algorithms are given for constructing trapezoidal diagrams of a set of line segments.

There are two main algorithmic methods used to construct the algorithms, one of them is randomized incremental construction and the other is random sampling. In random sampling, we take a random subset of a set of points $S \subset R$. It turns out that with high probability the convex hull of S splits R into small pieces, which can be exploited in an algorithmic approach. If we take a large subset R and recursively compute the convex hull then the points $R-S$ are expected to induce "local" changes which can be computed rather quickly. The random sampling approach gives rise to algorithms for halfspace-range reporting in the paper.

Randomized incremental construction is a generic algorithmic method for computing a set of ranges from a given set of objects. Objects can be points, line segments, halfspaces or balls. Each range is defined by a constant number of objects from S. For computing convex hulls in the place, the objects are points and ranges are halfplanes. In this case, a halfplane is bounded by a line through a pair of points. The construction problem determines all ranges F

[1] All time bounds are expected time

among all ranges defined by objects in S such that $F \cap s = \emptyset$ for all $s \in S$. In other words, F gives us a nice subdivision of the space with respect to S.

The construction algorithm adds objects from S to a set R in random order and maintains a set of ranges. In order to do this efficiently a conflict graph is maintained, which is a bipartite graph with vertex set $S \cup R$ and whose each edge set represents an intersection between an object and a region.[2] At k^{th} step of the algorithm, k objects from S have been added to R and the conflict graph contains all intersections between objects in $S - R$ and F so far.[3]

The conflict graph is used to convert a potentially slow incremental insertion sort like algorithm to a fast quicksort-like algorithm. The main disadvantage of insertion sort is that insertion is expensive. If it were possible, like heapsort, to insert elements in place in $O(logn)$ time, then insertion sort would be optimal. In fact, there is a simple way to achieve this. Let each vertex of the conflict graph be elements of input set and intervals of numbers. Each number initially intersects with the whole region. When adding k^{th} number all we have to do is to look at the region where x is and then partition those numbers in the region around x just like in quicksort. Essentially we have achieved the same algorithm as randomized quicksort which has expected $O(nlogn)$ time, but we are able to do it one by one. The generalized form of this approach allows for an object not yet added to intersect with multiple regions – unlike quicksort. However, the working principle is the same. Since we confidently know which regions we should look at, we can split the regions in question and then distribute the objects in the region into split regions, updating the conflict graph. The paper proves that if a region is bounded by a constant number of objects and if it also meets some additional criteria [4], then the expected number of operations will be quite favorable as in the case of line segment intersections. The proofs are somewhat involved and will not aid much to the purpose of this project report.

2 Goals

The goal of the project is to implement the randomized incremental construction in Ocaml language, possibly investing in a clean, modular and generic code. The language is quite adequate for computational geometry research since it combines functional and imperative aspects in an efficient manner, both of which are used in geometric algorithms. All geometric primitives and data structures will be implemented from scratch.

Secondly, an instance of the generic algorithm will be demonstrated. For this algorithm I have chosen the line segment intersection algorithm. Note that this algorithm is different from the randomized incremental algorithm described in our textbook.

3 Design

There are some aspects of the design that should be mentioned. First, the randomized incremental construction is implemented as a functor that takes a

[2]Therefore conflict graph is bipartite.

[3]In the paper, formal definitions are given for each version of F set and rigorous proofs are given for bounds.

[4]such as how fast the intersections and similar properties of a region can be determined

module with specific code and produces the particular construction code. A functor is simply a function that goes from a program module to a program module. One gives a module signature when defining the functor, therefore capturing the abstract properties of the input module. This will be used to realize the generic method in its intended mathematical formulation.

As input to this functor, there is a module for trapezoidal map construction that takes a set of line segments and incrementally builds a trapezoidal map depicting the spatial relationships between the segments. This should be thought of as the entry points from the general algorithm.

The required data structures such as graph and trapezoidal maps are implemented as modules which can hopefully be developed and tested separately from the main program. Interestingly, the hardest part has been implementing these data structures and geometric primitives.

For showing the generality of the functor, a sorting routine as described in this report should be developed as well.

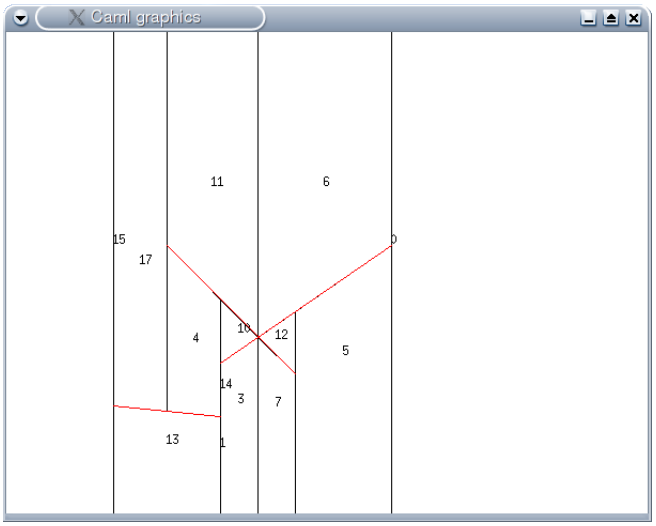

Figure 1: Trapezoidal map of three line segments

4 Program

In this section, I will describe each program module summarily. The package name of the program is known as "lasvegas-geom" referring to the necessarily random nature of the program. Figure 1 depicts the graphical output of the program for the input of array of line segments:

```
[| ((100.,100.),(200.,90.));((150.,250.),(270.,130.));
((200.,140.),(360.,250.))  |]
```

4.1 Randomized Incremental Construction

The construction functor determines types and the basic functions for construction. As seen, we demand types for the object, regions and a region set data

structure indexed by integers.

Two other important functions are initialization of the region set and a function to add object to the region set making use of conflict graph. The implementation is quite straightforward at the moment. It simply adds the objects in random order from input array using the given functions.

module Randomizedincr

```
(*
**
** ocaml module Randomizedincr
**
** Description: Generic randomized incremental construction algorithm
**
** Author: Eray Ozkural (exa) <erayo@cs.bilkent.edu.tr>, (C) 2003
**
** Copyright: See COPYING file that comes with this distribution
**
*)

open Printf

module type GeomIntxn =
sig
  type obj

  type region
  type region_set (*indexed by integers*)

  val init_regionset : unit -> region_set
  val add_obj : obj -> int -> Conflictgraph.t -> obj array -> region_set -> unit

  val print_obj :  obj -> unit
end

module Make = functor (Geom: GeomIntxn) ->
struct
  let random_permute n =
    let a = Array.init n (function i -> i)
    and swap a i j = let t = a.(i) in a.(i) <- a.(j); a.(j) <- t
    in
      Random.self_init ();
      for i=1 to n-1 do
        swap a (Random.int i) i
      done;
      a
  let construct s =
    let n = Array.length s
    in
    let p = random_permute n
    and g = Conflictgraph.make s
    and f = Geom.init_regionset ()
    in
      Array.iter (fun x->printf "%d " x) p;
      for i=0 to n-1 do
        let j = p.(i) in
```

```
49            Printf.printf "adding object %d :" j;
50            Geom.print_obj s.(j); Printf.printf "\n";
51            Geom.add_obj s.(j) j g s f;
52        done;
53        f
54  end
```

Since I didn't have a bipartite graph, I used two directed graphs to implement the conflict graph easily. As a matter of fact, this should grow to be a bipartite graph implementation because the directed graph calls tend to be error prone. This code initializes the graph with the required edges to denote the whole space and each object intersecting with it, which is a necessary condition.

module Conflictgraph

```
1
2   (* conflict graph type
3    * regions is a map obj -> a set of regions
4    * objs is a map region -> a set of regions
5    *)
6
7   type t = { regions: Digraph.digraph; objects: Digraph.digraph }
8
9   (* get conflicting regions of an object *)
10  let regions_of cg oix = Digraph.adj cg.regions oix
11
12  (* get conflicting objects of a region *)
13  let objects_of cg rix = Digraph.adj cg.objects rix
14
15  (* add a conflict between an object and a region *)
16  let add cg oix rix =
17    Digraph.add cg.regions (oix, rix);
18    Digraph.add cg.objects (rix, oix)
19
20  (* remove conflict *)
21  let remove cg oix rix =
22    Digraph.remove cg.regions (oix, rix);
23    Digraph.remove cg.objects (rix, oix)
24
25  (* initialize a conflict graph,
26     assume one region 0 that represents whole space *)
27  let make s =
28    let c = { regions = Digraph.make (); objects = Digraph.make ()} in
29    let add i x = add c i 0
30    in
31      Array.iteri add s;
32      c
```

4.2 Line segment intersections

This module details the input to the functor Randomizedincr.Make which outputs the complete trapezoidal map construction algorithm. The code here deals with maintaining the incremental data structures and calling appropriate primitive and structural routines. The bulk of the code here is concerned with conflict

graph and the abstract form of the construction procedure which may be moved inside the functor. I imagine there could be 2 or 3 more functions to be specified in the input module.

The addition algorithm performs all the steps specified in the paper. It splits each region that the added object intersects with updating the conflict graph. Then it merges those regions that can be merged. [5]

Figure 2 illustrates how the conflict graph is used when adding an object from $S - R$ to F. Conflict graph keeps track of intersections between objects in $S - R$ [6] and F_k [7]. The figure shows a subgraph of conflict graph. in the graph shows how an object oix is added from $S - R$ to R. The neighbors of oix are regions that oix intersects with. The bold line shows one of the regions rix that oix intersects with. Naturally oix is a line segment and rix is a trapezoid here. Once the object oix is added the trapezoidal diagram and conflict graph must be updated to reflect the situation. oix is going to split those regions that it intersects with. The conflict graph will be updated such that objects that intersect with regions split by oix will be partitioned to new regions. And of course both oix and the regions it has split will be removed from the graph. For instance, we see that rix has edges to $o1, oix, o3, o4$. When rix is split it is eliminated from the graph leaving its place to two child trapezoids $c1$ and $c2$ which are added to conflict graph. Since rix will be gone, its edges must be carried over to $c1$ and $c2$. Each object intersecting with rix, except oix which will be deleted, is subject to intersection tests to determine the new edges in conflict graph. The resulting new edges in our hypothetical conflict graph are depicted with dashed lines.

4.2.1 Reporting intersections

Reporting intersections is trivial. In the trapezoid split code, we make explicit line segment intersection tests while splitting a trapezoid with a line segment. With a single imperative, the intersections can be accumulated in an output buffer. I have added a print routine to report the intersections when detected as an example in `Trapezoid.split`.

module Linesegintxn

```
1  (*
2  **
3  ** ocaml module Linesegintxn
4  **
5  ** Description: Computes the trapezoidal map of a set of line segments
6  **
7  ** Author: Eray Ozkural (exa) <erayo@cs.bilkent.edu.tr>, (C) 2003
8  **
9  ** Copyright: See COPYING file that comes with this distribution
10 **
11 *)
12
13 open Lineseg
14 open Printf
15
```

[5] I have designed a nice algorithm for this that was not addressed in the paper

[6] Those objects which have not yet been added to R

[7] Current set of regions in region set

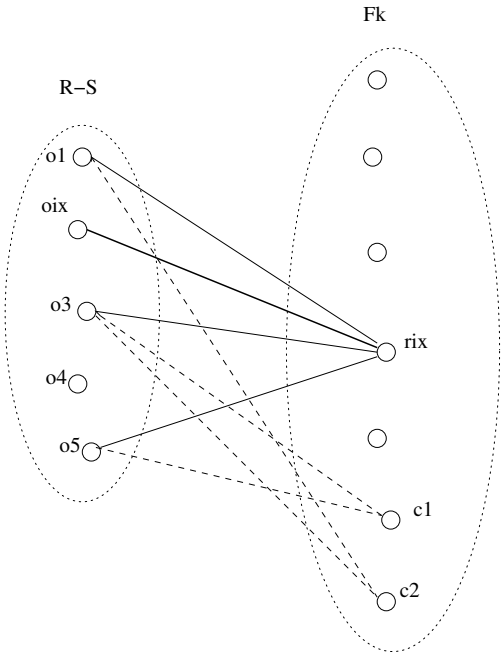

Figure 2: Adding an object to F_k using the conflict graph

```
let wait_key () =
  let g = Graphics.wait_next_event [Graphics.Button_down] in
    flush_all ()


module LineSegIntxn =
struct
  type obj = lineseg
  type region = Trapezoid.trapezoid
  type region_set = Trapdiag.trapdiag

  let init_regionset () =
    let t = Trapdiag.empty_trapdiag () in
    let i = Trapdiag.add_trap Trapezoid.empty_trap t in
      t

  let intersect ls trap = Trapezoid.intersect trap ls

(*
  let partition objects regions =        (* partition objects over regions *)
*)

  let sort_traps td l =
    let bnd x = (Trapezoid.test_bnd (Trapdiag.get_trap td x).Trapezoid.left
                   (fun l -> l) 0.) in
      List.sort (fun x y -> int_of_float (2. *. ((bnd x) -. (bnd y))) ) l

  (* add object with index oix in set s and conflict graph cg to region set
```

```
44      f *)
45    let add_obj obj oix cg s f =
46      let regions = Conflictgraph.regions_of cg oix in
47      let split rix =                (* split region with given object *)
48        printf "processing region: %d\n" rix;
49        let trap = Trapdiag.get_trap f rix in
50        let conflicting_objs = Conflictgraph.objects_of cg rix in
51          begin
52            (* remove conflict of new object with this region *)
53            Conflictgraph.remove cg oix rix;
54            let children = Trapezoid.split trap obj in (* split trap *)
55            let childrenix = Trapdiag.add_children f children
56            and objs_to_split = List.filter ((<>) oix) conflicting_objs
57            and intersecting_children = ref [] in
58              printf "childrenix: ";
59              List.iter (printf "%d ") childrenix; printf "\n";
60              Printf.printf "%d objects to split \n" (List.length objs_to_split);
61              List.iter
62                (fun ix ->
63                   (* remove conflict with split region *)
64                   Conflictgraph.remove cg ix rix;
65                   printf "split obj %d intersecting region %d:" ix rix;
66                   let intx_children =
67                     List.filter (fun cix -> intersect s.(ix)
68                                   (Trapdiag.get_trap f cix)) childrenix in
69                     (* update conflict graph *)
70                     List.iter (printf "%d ") intx_children; printf "\n";
71                     List.iter (fun cix -> Conflictgraph.add cg ix cix)
72                       intx_children
73                ) objs_to_split;
74              flush_all ();
75              Graphics.clear_graph ();
76              Graphics.set_color Graphics.black;
77              Trapdiag.draw f;
78              Graphics.set_color Graphics.blue;
79              List.iter Trapezoid.draw (Trapdiag.children_list children);
80              wait_key();
81              childrenix
82          end
83      and merge_traps unsorted_traps =
84        let td = f in
85        let rec iter_pairs f l = match l with
86            a::b::tail ->
87              let r = f a b in
88                ( match r with
89                      Some t -> iter_pairs f (tail)
90                    | None -> iter_pairs f (b::tail) )
91          | [a] -> ()
92          | [] -> () in
93        let s_traps = sort_traps td unsorted_traps in
94        let (upper,lower) = List.partition
95                              (fun ix -> let t = Trapdiag.get_trap td ix in
96                                 t.Trapezoid.lower=Trapezoid.Closed obj) s_traps
97        and process tnix tnpix =
```

```
98          let tn = Trapdiag.get_trap td tnix
99          and tnp = Trapdiag.get_trap td tnpix in
100           Graphics.set_color Graphics.blue;
101           Trapezoid.draw tn;
102           Graphics.set_color Graphics.red;
103           Trapezoid.draw tnp;
104           printf "processing traps %d %d\n" tnix tnpix;
105           printf "trap %d's left: " tnix;
106           Lineseg.print (Trapezoid.left_seg tn); printf "\n";
107           wait_key ();
108           Graphics.set_color Graphics.black;
109           Trapezoid.draw tn; Trapezoid.draw tnp;
110           if (tn.Trapezoid.upper)=(tnp.Trapezoid.upper)
111             && (tn.Trapezoid.lower)=(tnp.Trapezoid.lower) then
112               let t = { Trapezoid.left = tn.Trapezoid.left;
113                         Trapezoid.upper = tn.Trapezoid.upper;
114                         Trapezoid.right = tnp.Trapezoid.right;
115                         Trapezoid.lower = tn.Trapezoid.lower;
116                         Trapezoid.tl = tn.Trapezoid.tl;
117                         Trapezoid.tr = tnp.Trapezoid.tr;
118                         Trapezoid.br = tnp.Trapezoid.br;
119                         Trapezoid.bl = tn.Trapezoid.bl } in
120               let r1 = Trapdiag.remove_trap td tnix
121               and r2 = Trapdiag.remove_trap td tnpix
122               and tix = Trapdiag.add_trap t td in
123                 printf "MERGING traps %d %d\n" tnix tnpix;
124                 let objs1 = Conflictgraph.objects_of cg tnix
125                 and objs2 = Conflictgraph.objects_of cg tnpix in
126                   List.iter (fun x -> Conflictgraph.remove cg x tnix) objs1;
127                   List.iter (fun x -> Conflictgraph.remove cg x tnpix) objs2;
128                   List.iter (fun x -> Conflictgraph.add cg x tix )
129                     (objs1 @ objs2);
130                   Some tix
131           else
132             None
133       in
134         printf "*** Merging traps: ";
135         flush_all ();
136         Graphics.clear_graph ();
137         Graphics.set_color Graphics.black;
138         Trapdiag.draw f;
139         iter_pairs process upper;
140         iter_pairs process lower
141     in
142       printf "conflicting regions: ";
143       List.iter (fun x -> printf "%d:" x ) regions;
144       printf "\n";
145       (* split regions with given object *)
146       let children = List.concat (List.map split regions)
147       and traps = List.map (Trapdiag.remove_trap f) regions in
148         (* and finally merge traps *)
149         merge_traps children
150
151   let print_obj ls = Lineseg.print ls
```

```
152 end
153
154
155 module ConsTrapDiag = Randomizedincr.Make (LineSegIntxn)
156
```

4.3 Geometric primitives and structures

Lineseg module provides for abstraction of line segment and a lot of geometric primitives required. Of independent interest is a point-line classification routine and routines for computing intersection of two line segments, and a line segment and a horizontal/vertical line.

The following module defines a trapezoid record for the trapezoidal map that is bounded by two line segments from below and above and two vertical lines from sides. The bounds are potentially open. It also maintains four corner points. The hardest algorithm in the project is also here, computing how to split a given trapezoid with a line segment! A trapezoid can be split into four sub trapezoids, however the computation is not very straightforward. Also required was a routine to detect the intersection of a trapezoid with a line segment, that in turn requiring a routine to detect if a point is inside the trapezoid.

Trapezoidal map is constructed simply from a dynamic array of trapezoids and an adjacency graph[8]. The removed nodes go into a dead list, and are reused in file organization fashion. I am trying to figure out if the adjacency graph is really necessary and if so how it should be accessed. It has taken a lot of time for me to come up with this design, as I did not consider DCEL handling these specific trapezoids in a clean way although it would be the perfect choice if all I needed was a planar graph.

4.4 General purpose data structures

All the data structures used have been designed and implemented from the very scratch for this project. I have not been able to find high quality imperative structures that were essential for realizing the running time bounds.

I have programmed a dynamic array code using open table implementation. This is a quite useful code and together with a few interfaces it could be contributed to the Ocaml standard library.

After writing a dynamic array, the adjacency list implementation for graph data structure becomes a straightforward task. The module interface is quite fat, but it is worth the bother. Writing module interfaces in separate files ameliorate code reuse. As can be seen, I have concentrated on the primitive operations that allow for typical efficient graph algorithms to be written.

I have implemented each adjacency by an ocaml list. The adjacencies are coupled together using the dynarray type. Since this functional structure takes care of sharing values it should be as efficient as possible, however I have not yet seen the performance impact of this. If there is a performance impact, dynarray can be used instead of the list – which was the first implementation of this module. And finally, there is a small adaptor module for undirected graphs

CLI code and test routines, etc. have been omitted from this section.

[8]Which is not used yet in the code

5 Conclusion

I have reached my goal of being able to program the randomized incremental construction in a generic way. The program, although short, has accumulated a lot of programming work. According to SLOCCOUNT by David Wheeler it has 740 physical source code lines of ML code, totaling to 2.4 months of development time. [9] The geometric primitives and structures are also well satisfactory. The trapezoidal map construction algorithm is implemented effectively and it is accompanied with algorithm animation routines which I have used for graphical debugging.

[9] It took three weeks of very hard work for me