Local Monotonicity Reconstruction

Michael Saks
Rutgers University

C. Seshadhri
IBM Almaden

Overview

- Introduce a class of algorithmic problems:

  Local Property Reconstruction

  Distributed Property Reconstruction

  Parallel Property Reconstruction

  extending framework of

  program self-correction,

  robust property testing

  locally decodable codes)

- An interesting example: Monotonicity

Data Sets

Data set = function $f : \Gamma \rightarrow V$

Γ = finite index set

V = value set

For us,

$\Gamma = [n]^d = \{1,\ldots,n\}^d$

V = nonnegative integers

f = d-dimensional array of nonnegative integers

Properties of data sets

Focus of this talk:

- Monotone: nondecreasing along every line

(Order preserving)

When d=1,

monotone = sorted

Distance between two data sets

dist(f,g) = fraction of domain where $f(x) \neq g(x)$

$\varepsilon(f) = d(f,P)$

= minimum of dist(f,g) for g satisfying P

Property Reconstruction

Setting:

Given f

- We expect f to satisfy P

  (e.g. we run algorithms on f that rely on P)

- but f might not satisfy P

  but f is close to P -- ε(f) is small

Reconstruction problem for P

Given function f,

produce reconstructed function g that:

- satisfies P

- is close to f:

  Error blow-up

  $d(f,g) / \varepsilon(f)$

  is not too large

What does it mean to produce g?

- Offline property reconstruction

  Input: function table for f

  Output: function table for g

- Local property reconstruction

  (which builds on

  Online property reconstruction

  (Ailon-Chazelle-Liu-Seshadhri))

Local property reconstruction 1

What we want: A local filter

Algorithm A with query access to function f

Input: domain element x

Output: g(x) (reconstructed function value)

- A may query f(y) for any y
- uses short random string s
  -- otherwise deterministic.

Key points:

- String s is the same for all queries.
- The reconstructed function g is fully determined by f and s

Local Property Reconstruction II

Goals:

- g has property P (no error)

- d(g,f) = O(ε(f))

  WHP over choices of random string s

- For each input x, A(x) runs quickly

  in particular only reads f(y) for a small number of y.

Local Property Reconstruction III

Motivation:

- Allows for online reconstruction with small auxiliary memory
- Allows for many autonomous clients to perform the same reconstruction

Local Property Reconstruction III

Inspirations and Connections:

- Online Property Reconstruction (Ailon-Chazelle-Liu-Seshadhri)
- Locally Decodable Codes and Program self-correction (Blum-Luby-Rubinfeld; Rubinfeld-Sudan; etc)
- Graph Coloring (Goldreich-Goldwasser-Ron)
- Monotonicity Testing (Dodis-Goldreich- Lehman-Raskhodnikova-Ron-Samorodnitsky; Goldreich-Goldwasser- Lehman-Ron-Samorodnitsky;Fischer;Fischer-Lehman-Newman-Raskhodnikova-Rubinfeld-Samorodnitsky;Ergun-Kannan-Kumar-Rubinfeld-Vishwanathan; etc)
- Tolerant Property Testing (Parnas, Ron, Rubinfeld)

Example: Local Decoding of Codes

f = boolean string of length n

Property = is a Code word of a
given error correcting code C

Reconstruction = Decoding to a close code word

Local filter= Local decoder

Key issue for general properties

Answers must be mutually consistent

Say that h is satisfactory if it satisfies P and is close to f.

- We want a satisfactory h
- There may be many satisfactory h

- If we look at a single query point x, the algorithm may answer h(x) for any satisfactory h

  (possibly many permissible answers)

- Global consistency requirement:

  For each random seed, the ensemble of query responses corresponds to a single satisfactory h.

Our results I

A local filter for monotonicity in dimension d such that:

- Time to compute $g(x)$ is $(\log n)^{O(d)}$ as
- $dist(f,g) = C_1(d)d(f,P)$ ($C_1(d) = 2^{(O(d^2))}$)
- Shared random string s has size $(d \log n)^{O(1)}$

(Builds on prior results on monotonicity testing and online reconstruction mentioned earlier)

Lower Bound. For some $B>0$,

For any local filter for monotonicity on domain $\{0,1\}^d$

if query time is at most 2^{Bd}

then error blow up is at least 2^{Bd}

Other Examples and an Invitation

Other examples of local property reconstruction:

Not many….

- Locally Decodable Codes
- Graph k-colorability (Implicit in Goldreich-Goldwasser-Ron)
- Being an expander (Kale, Peres, Seshadhri)

INVITATION

Remainder of Talk:

Overview of our filter construction for monotonicity

Preliminaries

A subset S of Γ is f-monotone

if f restricted to S is monotone.

For each x in Γ, A(x) must:

- Decide whether g(x) = f(x)
- If not , then determine g(x)

Accepted = { x : g(x) = f(x) }

Rejected = { x : g(x) ≠ f(x) }

In particular, Accepted must be f-monotone

Subproblem: Element Classification

- Classify each x in Γ as Accepted or Rejected
  - Accepted is f – monotone
  - Rejected is small: size size O(ε(f)|Γ|)

Need subroutine Classify(x).

Initial approach

- Construct a subroutine Classify as above

- Define g(x):

$$\text{Accepted}(x) = \{ y : y \leq x \text{ and } y \text{ Accepted}\}$$

$$g(x) = \max\{f(y) : y \text{ in Accepted}(x)\}$$

- Then:
  - g is monotone
  - g agrees with f on Accepted

Initial approach II

Failure of initial approach

$$Accepted(x) = \{ y : y \leq x \text{ and } y \text{ Accepted}\}$$

$$g(x) = \max\{f(y) : y \text{ in } Accepted(x)\}$$

Computing $g(x)$ is expensive:

it (apparently) requires

identifying all maximal y in $Accepted(x)$

Refined approach

Given function Classify

Define

Accepted*(x) = a small carefully chosen sample of Accepted(x)

g(x) = max{f(y) : y in Accepted*(x))}

Refined Approach II

g(x) = max{f(y) : y in Accepted*(x))}

Resulting g need not be monotone

To ensure monotonicity

we need samples associated to each point to be compatible:

For all x < y, Accepted*(x) << Accepted*(y)

(Each z in Accepted*(x) is less than some z' in Accepted*(y))

Refined Approach III

Summary:

Two routines:

Classify(x) which Accepts or Rejects

Accepted*(x) gives sample of Accepted elements ≤ x
so that Accepted*(x) << Accepted*(y)

Return g(x) = max{f(y) : y in Accepted*(x))}

Refined Approach IV.

On input x,

Return g(x) = max{f(y) : y in Accepted*(x))}

Challenges:

(1) For most x, want x in Accepted*(x)

so as to guarantee g(x)=f(x)

(2) Need that sets Accepted*(x) are pairwise compatible

Conflict between (1) and (2)

Constructing Classify

- Classify each x in Γ as Accepted or Rejected
  - Accepted is f – monotone
  - Rejected is small:

    size $O(d(f,P)\ |\Gamma|)$

A sufficient condition for f-monotonicity

A pair (x,y) in $\Gamma \times \Gamma$ is a violation if

$$x < y \text{ and } f(x) > f(y)$$

To guarantee that Accepted is f - monotone:

Rejected should hit all violations:

For each violation (x,y), x or y is Rejected

Classify: 1-dimensional case

$d=1$: $\Gamma=\{1,\ldots,n\}$

f is a linear array.

For x in Γ, and subinterval J of Γ:

violations$(x,J)=|\{y \text{ in } J : (x,y) \text{ is a violation}\}|$

Interval J is near x if $\text{dist}(J,x)<|J|$

Constructing a large f-monotone set I

The set Bad:

x in Bad if for some interval J near to x
x is in violation with at least half of J

Lemma.
1) Good=Γ - Bad is f-monotone
2) $|Bad| \leq 4\, d(f,P)\, |\Gamma|$.

Proof:
1) If (x,y) is a violation then one of them is Bad for the interval [x,y].
2) Omitted, but easy

Constructing a large f-monotone set II

Lemma.

- Good=Γ \ Bad is f-monotone
- |Bad| ≤ 4 d(f,P) |Γ| .

So we'd like to take:

Accepted=Good   Rejected = Bad

How do we classify x as Good or Bad?

- To determine is y is Good or Bad:

  For each interval J that is near to y,
  is y is in violation with half of J?

  Too slow…..

- There are $\Omega(n)$ intervals J near to y
- Counting violations of x with J takes time $\Omega(|J|)$.

Speeding up the computation

- Estimate number of violations of y with J by random sampling from J

  sample size polylog(n) is sufficient

  violations* (y,J) denotes the estimate

- Compute violations* (y,J) only for a carefully chosen set of test intervals

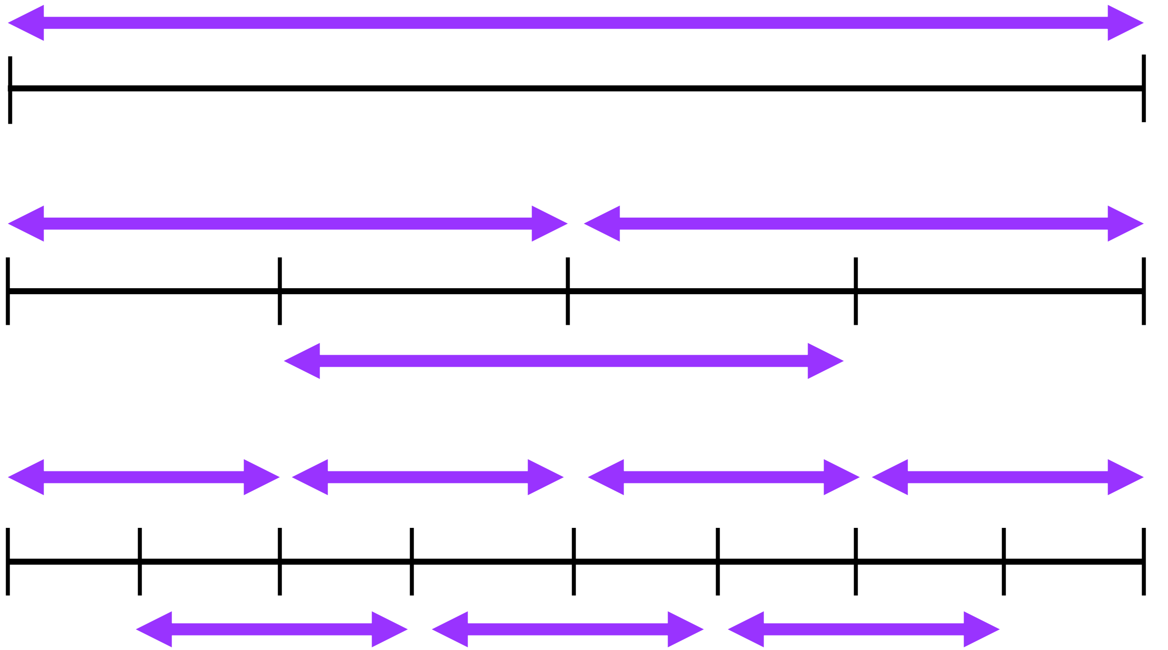
Set of Test intervals

Want set T of test intervals of [n] satisfying:

- Each x is near to O(log n) test intervals
- For any $x<y$, there is a test interval contained in [x,y] that is near to both x and y.

which ensures that WHP, for every violation x,y, at least one of them is rejected.

The Test Set T



Assume $n=|\Gamma|=2^k$

k layers of intervals

Layer j consists of $2^{k-j+1}-1$ intervals of size 2^j

Subroutine classify

To classify y

If for each J in T near to y

violations*(y,J) < .4 |J|

then y is Accepted

else y is Rejected

Where are we?

For $d=1$ have a subroutine Classify

- On input x,
  - Classify outputs Accepted or Rejected
  - Runs in time polylog(n)

- WHP
  - Accepted is f-monotone
  - $|\text{Rejected}| \le 10\, d(f,P)\, |\Gamma|$

Lift to higher d by recursion on dimension

Where are we? II

Now we need a fast function:

Accepted*(x):

returns a carefully chosen sample of Accepted elements ≤ x

Must satisfy:

for all $x < y$,

Accepted*(x) << Accepted*(y)

Constructing Accepted*(x), d=1

- Use the same test intervals.
  - For each test interval J construct a polylog(n) size sample Sample*(J)
  - First attempt: Take Accepted*(x) to be:

    union of Sample*(J) for J

    near to and ≤ x

But this may violate

Accepted*(x) << Accepted*(y)

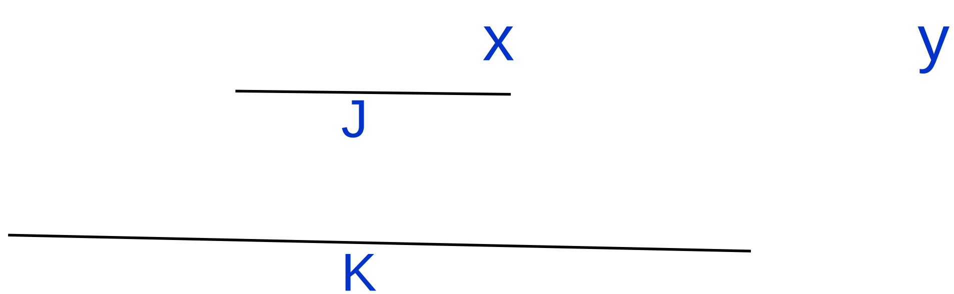
Constructing Accepted*(x), d=1 II

- First attempt: Take Accepted*(x) to be:

  union of Sample*(J) for J

  near to and ≤ x

Bad Scenario: x < y



x in Sample*(J)

but not Sample*(K)

Avoiding the bad scenario, d=1

Focus on the test intervals

- For a given test interval J, there are only
  O(log n) test intervals J'
  that can cause the bad scenario.

- For each such J',
  if the bad scenario happens
  then set Sample*(J) to be empty.

- Key point in analysis: can still ensure that g(x)=f(x) for "most" x.

Constructing Accepted*(x), d>1

Instead of O(n) test intervals,

have $O(n^d)$ test boxes

Construct Sample*(B) for each box B.

Identify similar bad scenario, but….

….. Setting Sample*(B) to be empty is too drastic.

Instead Sample*(B) is thinned out carefully

This is the hardest part of the paper:

technical (but not messy) algorithm and analysis

Further work

- Technical (but still interesting) gap:

  The g produced by our algorithm has

  $$d(g,f) \le C(d)\varepsilon(f)|\Gamma|$$

  - Upper bound on $C(d)$ is $\exp(d^2)$.
  - Lower bound on $C(d)$ $\exp(d)$

- Main question:

Are there other interesting properties with non-trivial local filters?

  - (Reconstructing expanders, Kale,Peres, Seshadhri, FOCS 08)