Sublinear time algorithms II

Ronitt Rubinfeld
MIT

FODSI Summer School August 2022

Plan

- Yesterday:
  - Diameter of point set
  - Estimate the degree of a graph
  - Estimate the number of connected components of a graph
  - Estimate Minimum Spanning Tree weight

- Today:
  - Sublinear algorithms from distributed algorithms
  - Sublinear algorithms from greedy algorithms
  - Property testing -- monotonicity

More specific plan

- Oracle reduction framework
- Implementing the oracle via simulating parallel algorithms in sublinear time
- Implementing the oracle via simulating greedy algorithms in sublinear time
- Property testing -- monotonicity

The oracle reduction framework [Parnas Ron]

Example problem: Vertex Cover

- Given graph G(V,E), a vertex cover (VC) C is a subset of V such that it “touches” every edge.

- What is minimum size of a vertex cover?
  - NP-complete
  - Poly time multiplicative 2-approximation based on relationship of VC and maximal matching

Approximation for VC

- Multiplicative?
  - VC of graph with no edges vs. graph with 1 edge
- Additive?
  - Need to allow some multiplicative error: Computationally hard to approximate to better than 1.36 factor
- Combination?
  - Def. y' is (α, ϵ)-estimate of y if
    $y \le y' \le \alpha \cdot y + \epsilon \cdot n$

Good for minimization problems

Def. y’ is (α, ϵ)-estimate of y
if $y \le y' \le \alpha \cdot y + \epsilon \cdot n$

Vertex cover approximation

- Can get CONSTANT TIME (α, ϵ)-estimate for vertex cover on sparse graphs!

How?

- Oracle reduction framework [Parnas Ron]
  - Construct “oracle” that tells you if node *u* in 2-approx vertex cover
  - Use oracle + standard sampling to estimate size of cover

But how do you implement the oracle?

Implementing the oracle – two approaches:

- Sequentially simulate computations of a fast distributed algorithm [Parnas Ron]

- Figure out what greedy maximal matching algorithm would do on u [Nguyen Onak]

Constructing oracles via distributed algorithms

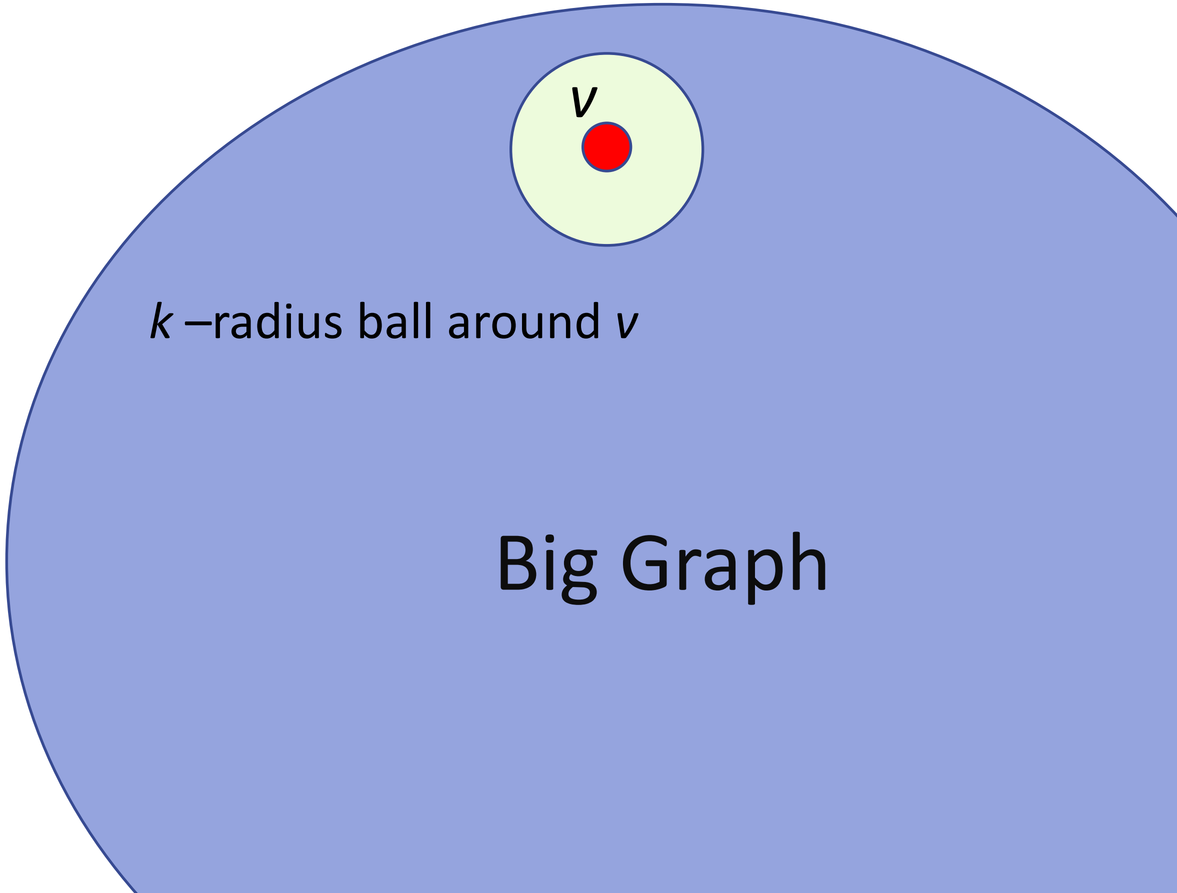
Distributed Algorithms: LOCAL model (simple version)

- Network
  - Processors
  - Links
  - (assume maximum degree is known to all)
- Communication round
  - Each node sends message to each neighbor
- Vertex Cover Problem:
  - Network graph = input graph
  - After k rounds, each node knows if it is in VC

LOCAL distributed algorithms give sublinear algorithms for oracles

[Parnas Ron]

- If there is a *k* round distributed algorithm for VC, then:
  - *v*'s output depends only on inputs (unique IDs, neighbors, randomness) and computations of *k*-radius ball around *v*
  - **Sequentially** read/simulate in Δ^k probes!
- How big is *k*?



How fast are distributed algorithms?

- Vertex cover: $O\left(\left(\frac{d}{\epsilon}\right)^{O(\log d/\epsilon)}\right)$ sequential time via [Kuhn Moscibroda Wattenhoffer]

- Lots and lots of very fast distributed algorithms!
    - Packing and covering problems, matching, maximal independent set, coloring,…

Oracle reduction framework via simulating distributed algorithms

Thm [Parnas Ron]: *t*-round distributed algorithm for vertex cover yields $d_{max}^{O(t)}$ sequential query approximation algorithm for vertex cover.

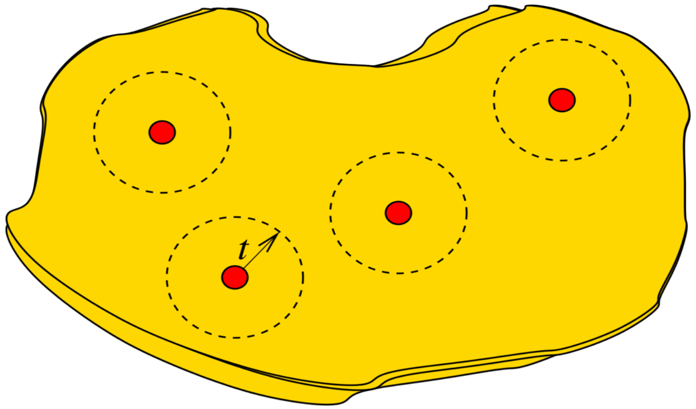
Estimation idea:

Sample vertices of graph

For each sampled vertex *v*,

simulate distributed algorithm to see if *v* is in VC

Output *(fraction in VC)·n*



Bounded degree graph G

Constructing Oracles via simulating greedy

Vertex Cover and Maximal Matching

- Maximal Matching:
  - $M \subseteq E$ is a matching if no node in in more than one edge.
  - M is a maximal matching if adding any edge violates the matching property

- Classic result: nodes of M are a pretty good Vertex Cover!

  (i.e., no more than twice value of optimal → Maximal matching gives good enough approximation)

Greedy algorithm for maximal matching

- Sequential Greedy Algorithm:
  - $M \leftarrow \emptyset$
  - For every edge (u,v)
    - If neither of u or v matched
      - Add (u,v) to M
  - Output M

Which order?

Rank!

- Why is M maximal?
  - If (u,v) not in M then either u or v already matched by earlier edge

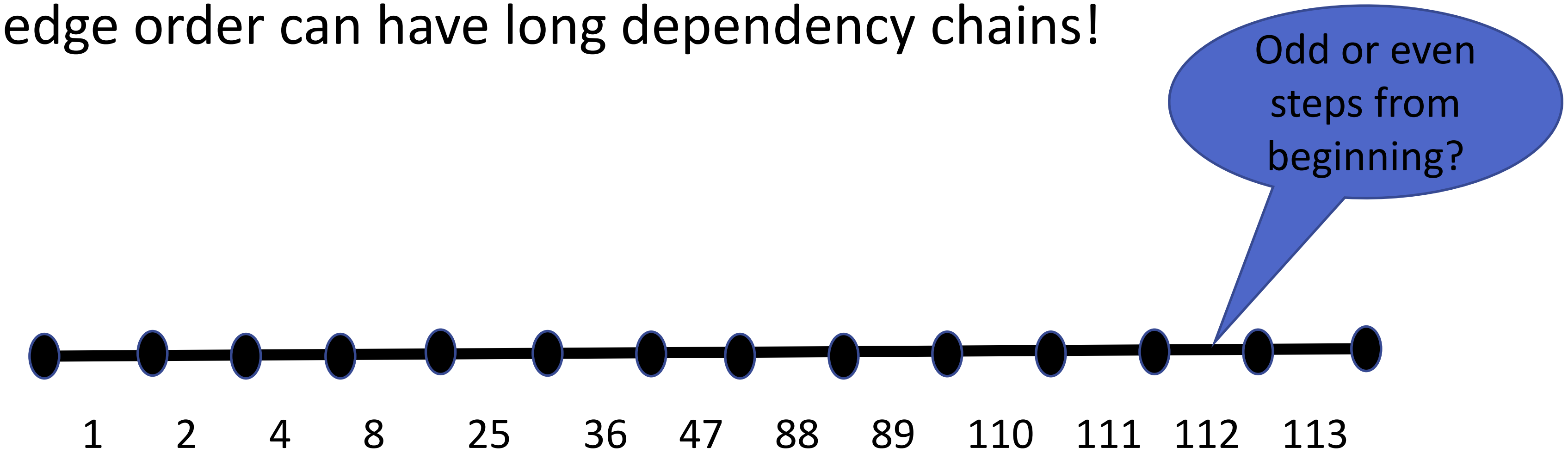
Why can local algorithms hope to simulate behavior of greedy?

- Easy case: If edge has smaller rank than all neighboring edges, greedy will put it into matching

Implementing the Oracle via Greedy

- To decide if edge e in matching:
  - Must know if adjacent edges that come *before* e in the ordering are in the matching
  - Do not need to know anything about edges coming *after*

- Arbitrary edge order can have long dependency chains!



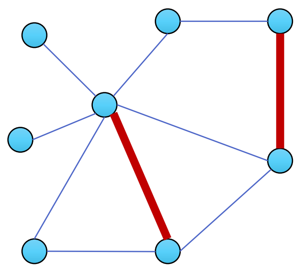
Breaking long dependency chains
[Nguyen Onak]

- Assign random ranks (ordering) to edges
  - Greedy works under any ordering
  - Important fact: random order has short dependency chains

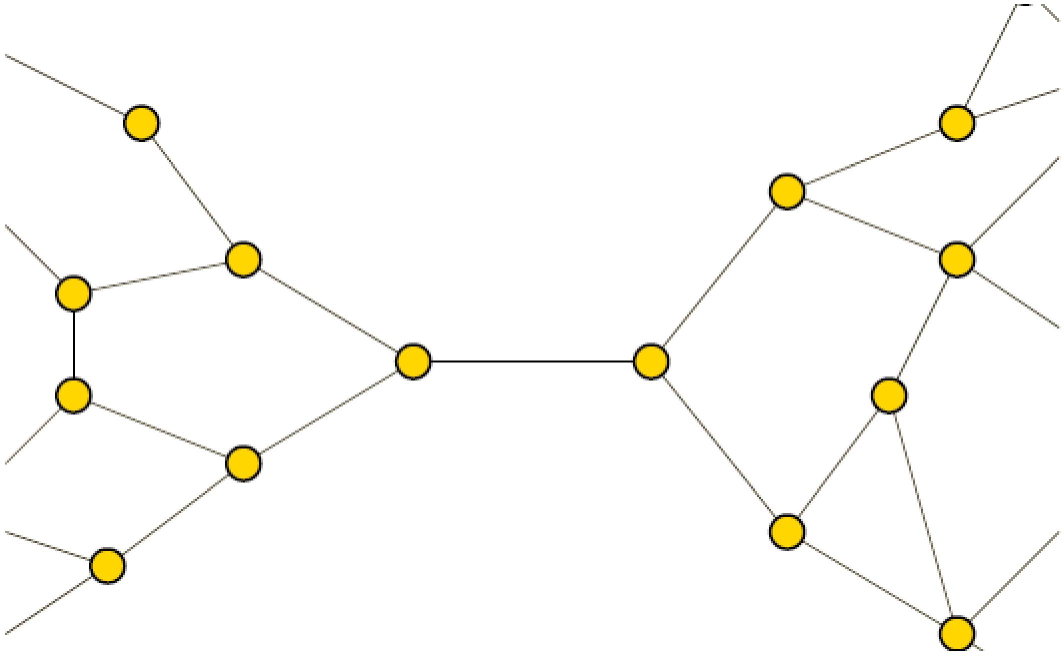
Implementing oracle $\mathcal{O}$

[Nguyen Onak]

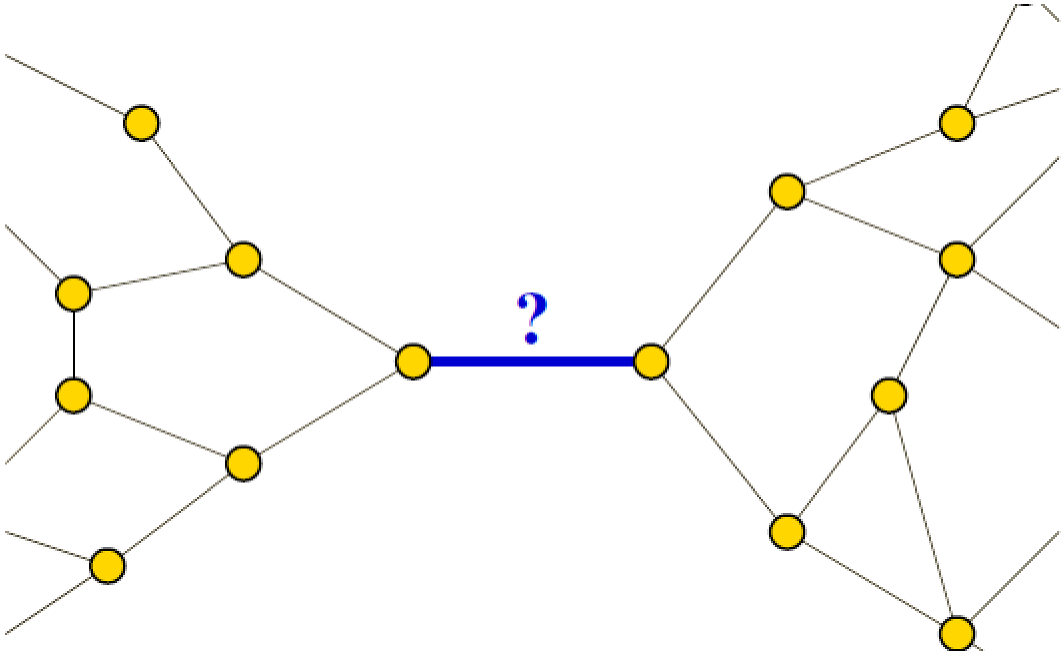
- Preprocessing:
  - assign random number $r_e \in [0,1]$ to each $e \in E$
- Oracle implementation:
  - Input: edge $e \in E$,
  - Output: is e in M?
  - Algorithm:
    - Find all the adjacent edges of e, e'∈E, such that $r_{e'} < r_e$
    - Recursively check if any in *M*
      - If any in the matching, output NO
      - If none are in the matching, output YES



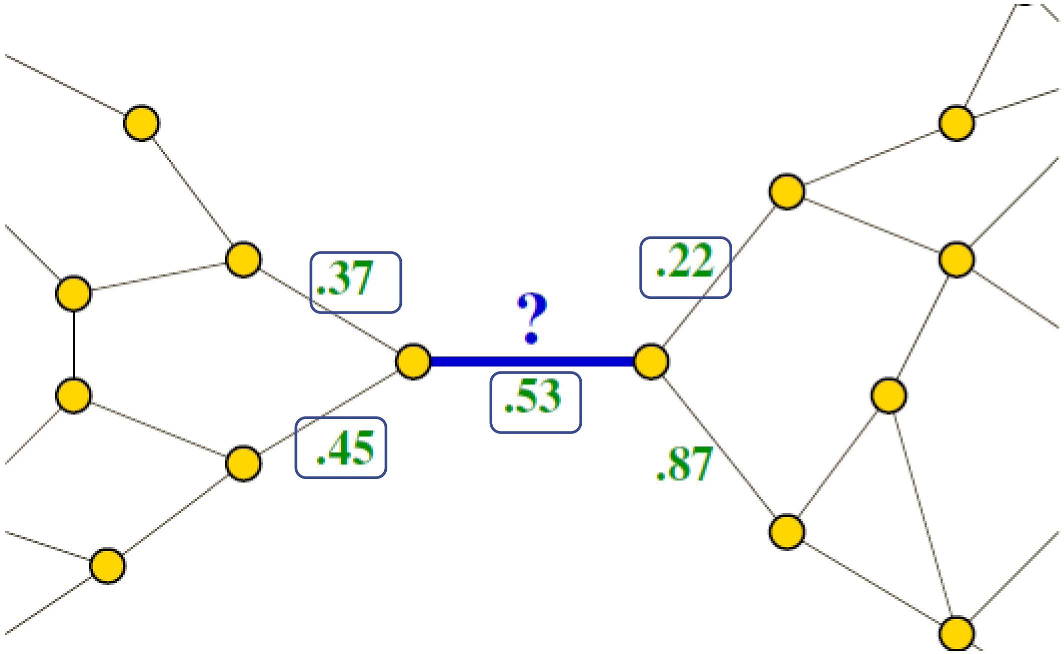
Example Run $\mathcal{O}$



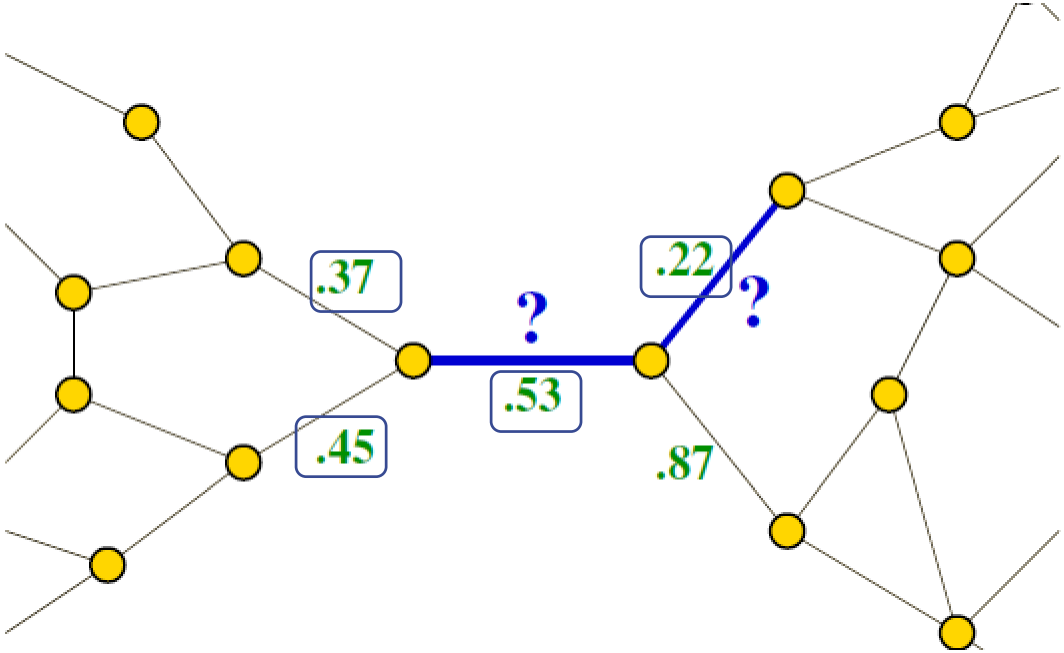
Example Run $\boldsymbol{\mathcal{O}}$ (cont.)



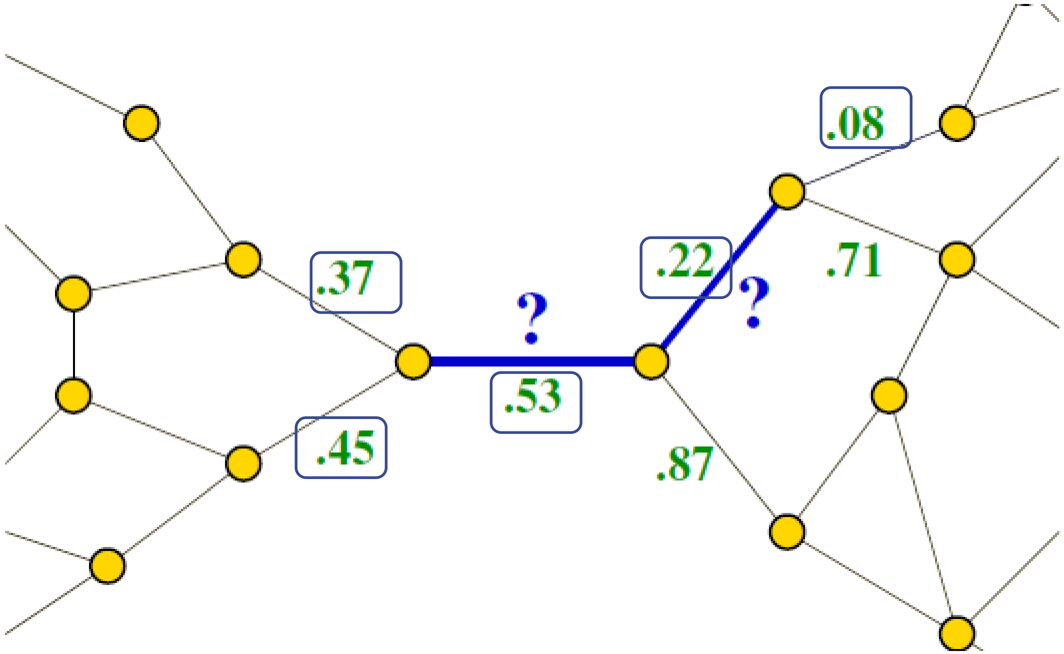
Example Run $\mathcal{O}$ (cont.)



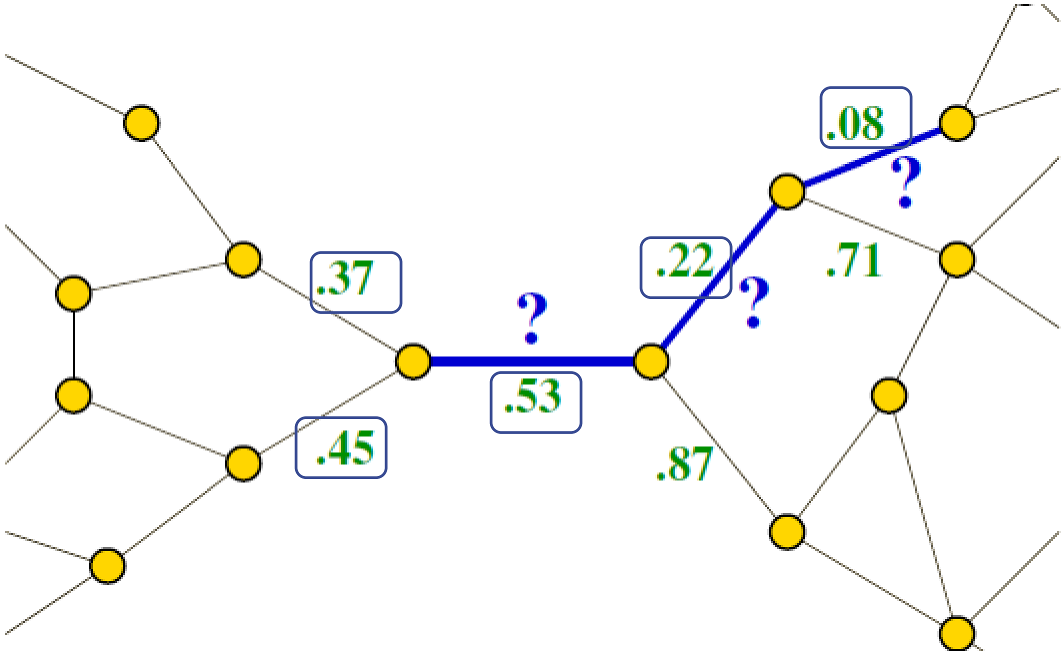
Example Run $\mathcal{O}$ (cont.)



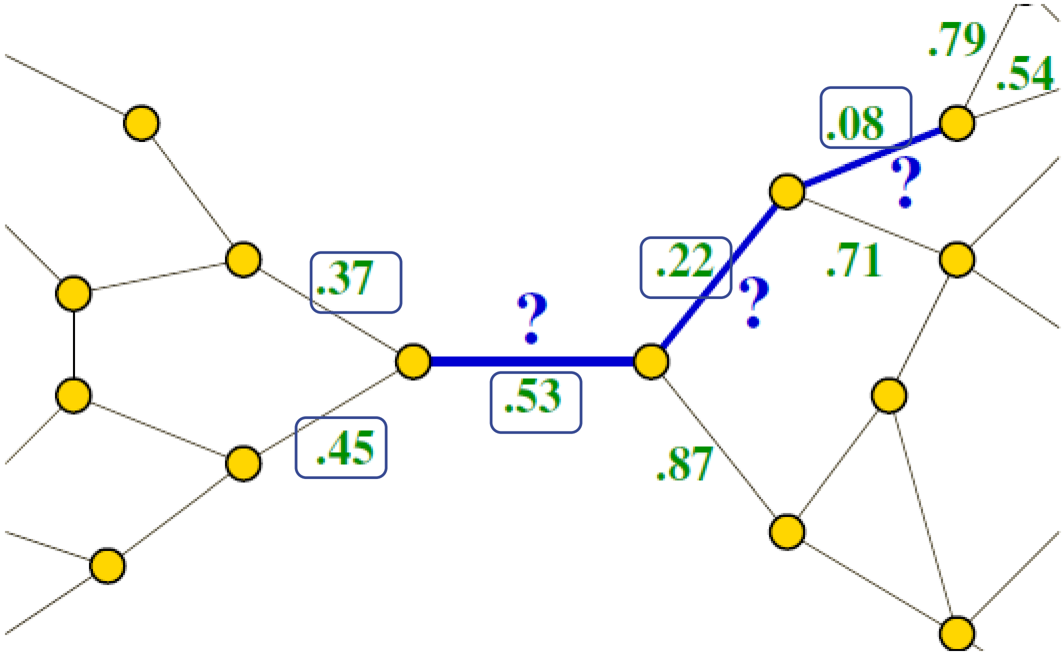
Example Run $\mathcal{O}$ (cont.)



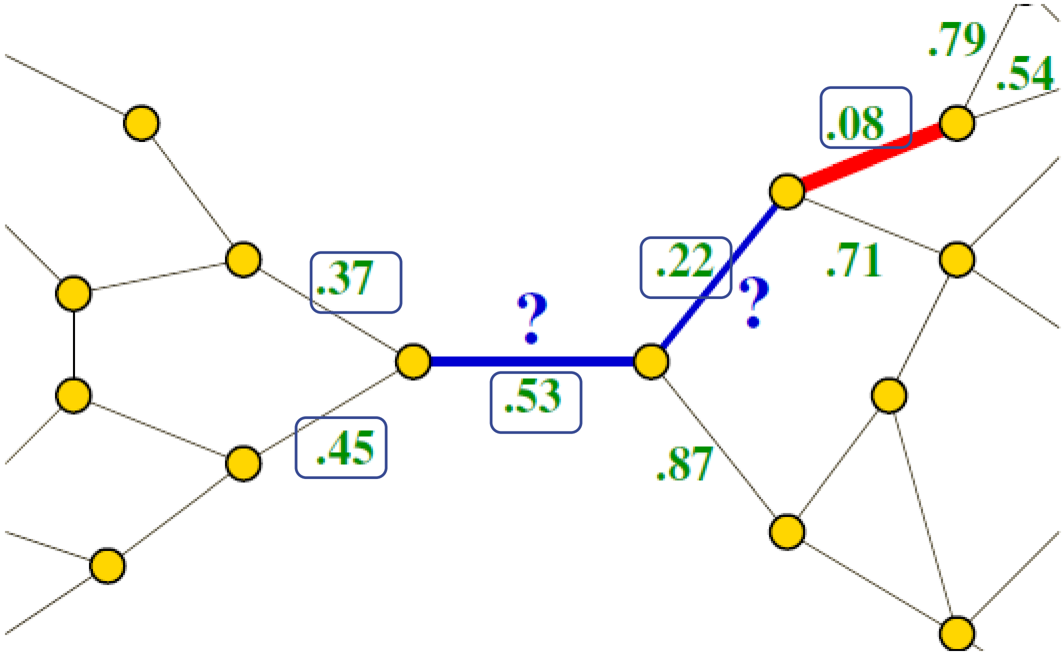
Example Run $\mathcal{O}$ (cont.)



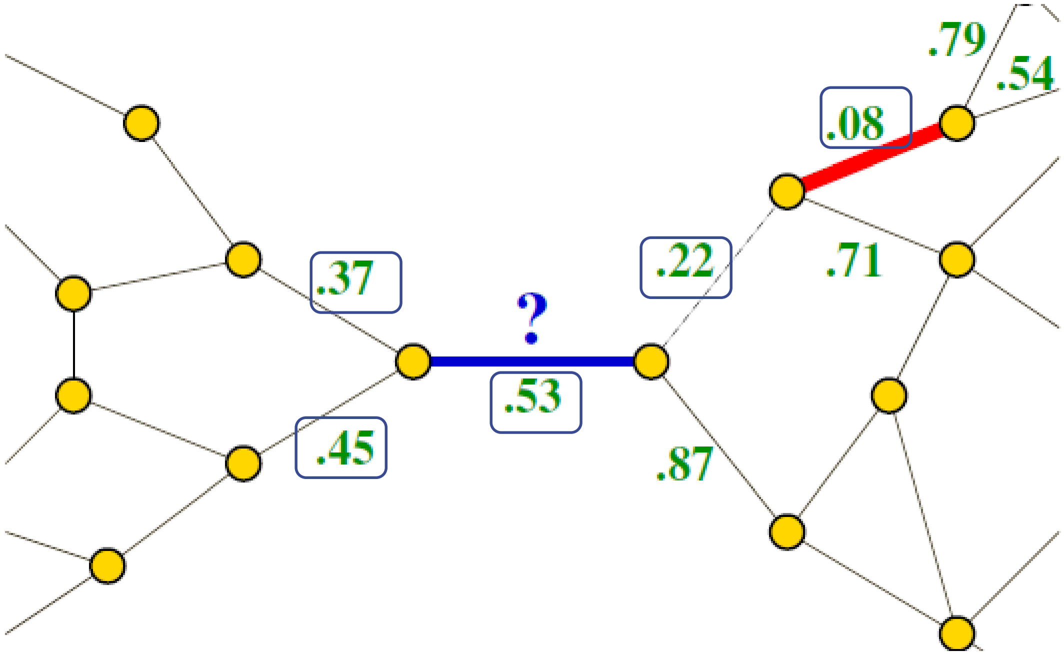
Example Run $\mathcal{O}$ (cont.)



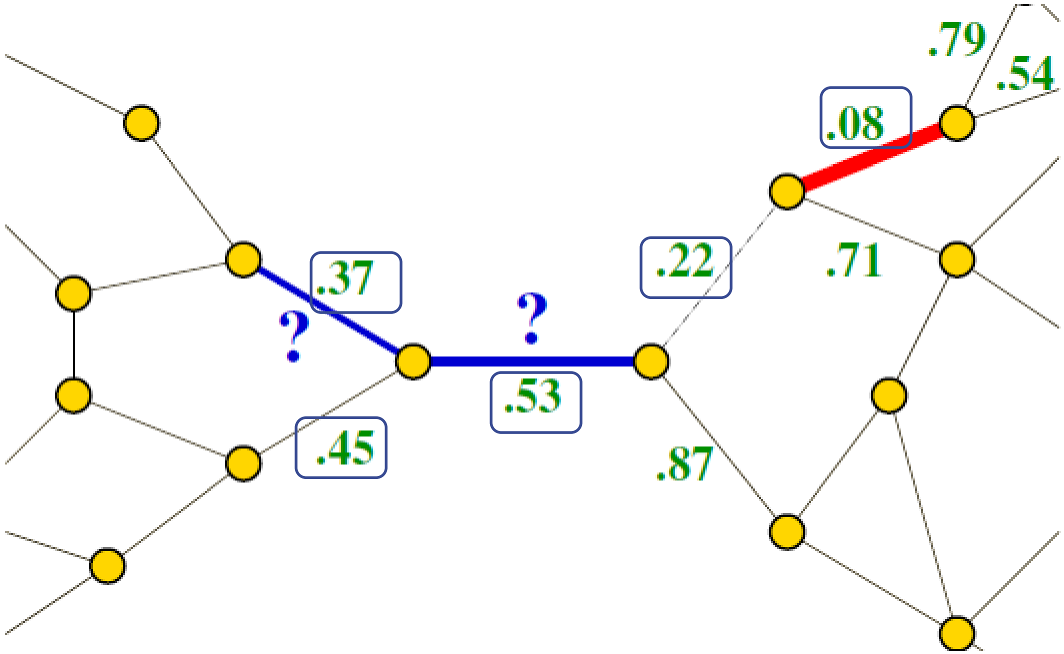
Example Run $\mathcal{O}$ (cont.)



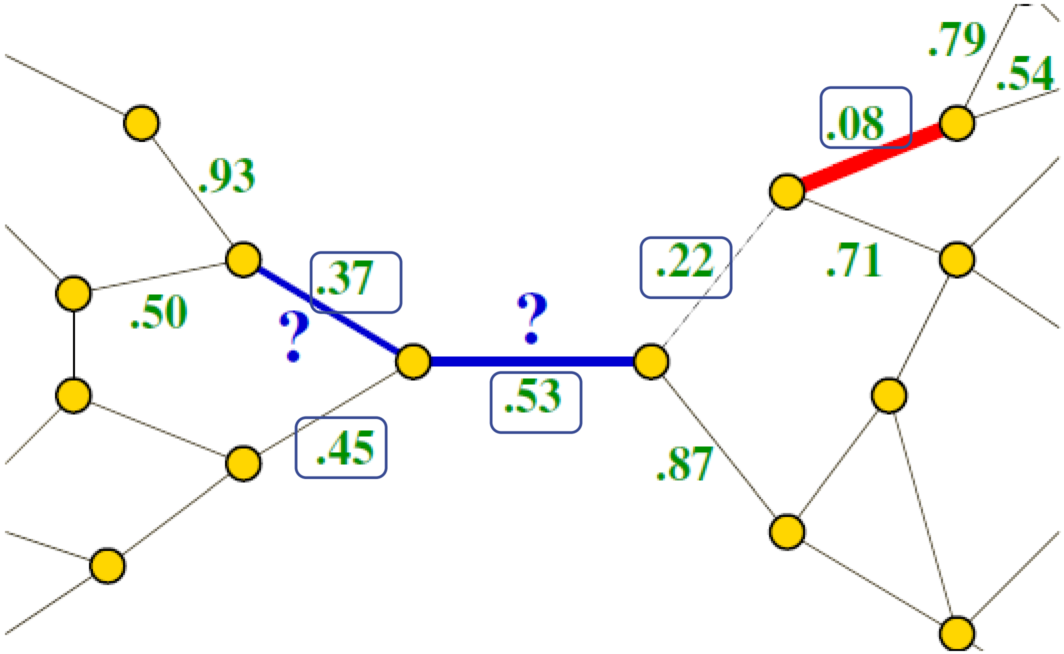
Example Run $\mathcal{O}$ (cont.)



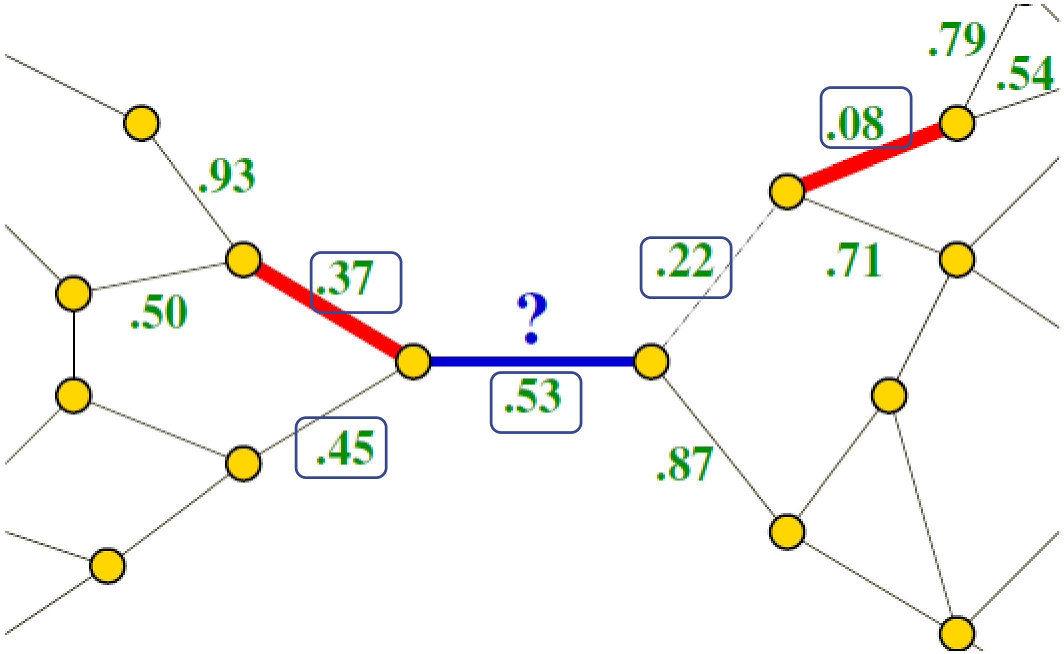
Example Run $\mathcal{O}$ (cont.)



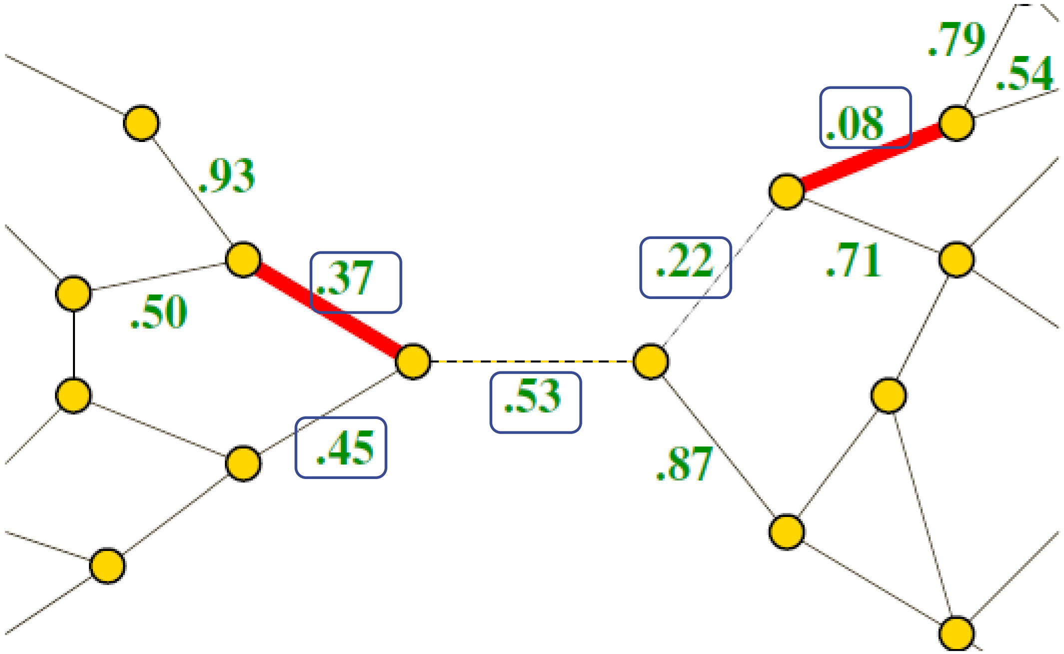
Example Run $\mathcal{O}$ (cont.)



Example Run $\mathcal{O}$ (cont.)



Example Run $\mathcal{O}$ (cont.)



Correctness

- This algorithm simulates run of classical greedy algorithm
  - Greedy works under any ordering of edges

- Outputs estimate t such that

$$MM(G) \leq t \leq MM(G) + \epsilon n$$

where MM(G) is size of some maximal matching

Complexity

- Claim: Expected number queries to graph per oracle query is $2^{O(d)}$
  - Total complexity is $2^{O(d)} / \epsilon^2$
  - Main idea:
    - Bound probability a path of length k explored:
      - Ranks must decrease along the path
      - So probability $\leq 1/(k)!$

Complexity

- Claim: Expected number queries to graph per oracle query is $2^{O(d)}$
- Proof:
  - Pr[given path of length k explored] $\leq 1/(k)!$
  - Number of neighbors at distance k $\leq d^k$
  - E[Number of nbrs explored at dist k] $\leq d^k/(k)!$
  - E[number of explored nodes] $\leq \sum_{k=0}^{\infty} d^k/(k)! \leq e^d/d$
  - E[query complexity] = $O(d)\, e^d/d$

    $= 2^{O(d)}$

Better Complexity for VC

- Always recurse on least ranked edge first
  - Heuristic suggested by [Nguyen Onak]
  - Yields time nearly linear in degree [Yoshida Yamamoto Ito][Onak Ron Rosen R.] [Behnezhad]

Further work

- More complicated arguments for maximum matching, set cover, positive LP… (and lots more)

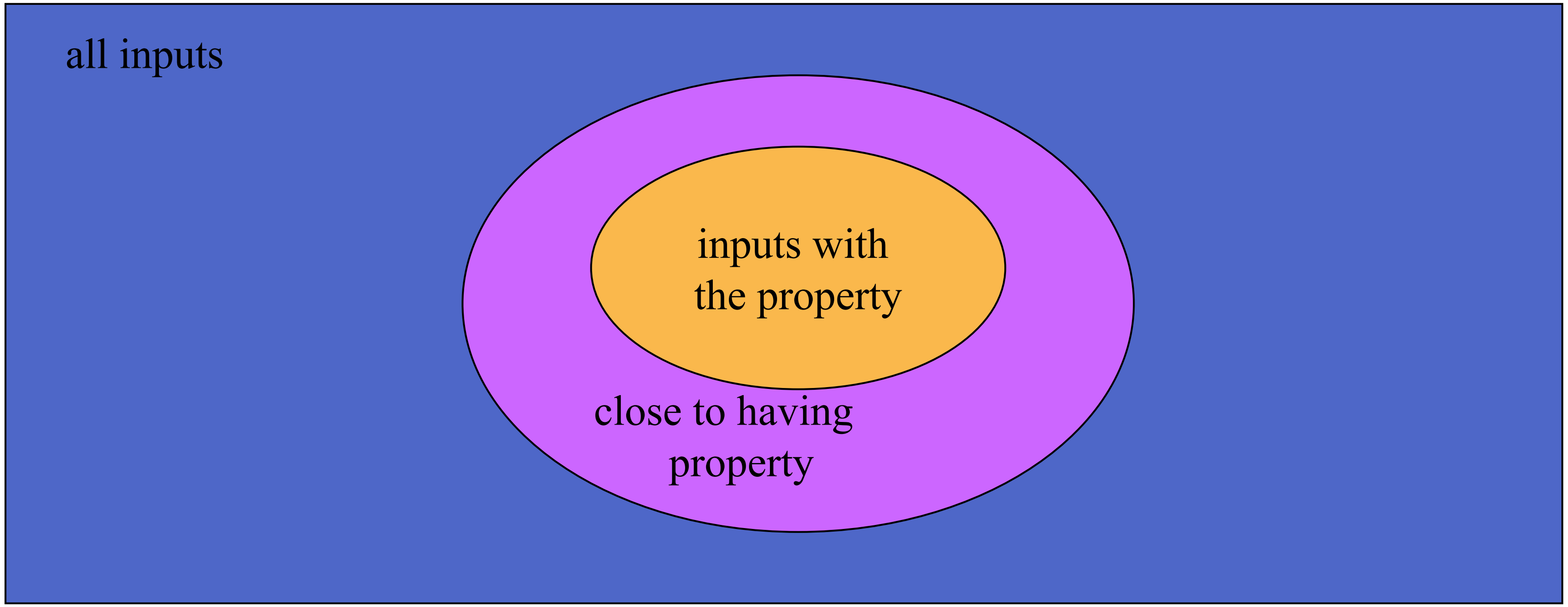
> Can dependence be made poly in average degree?

- Even better results for some of these problems on hyperfinite graphs [Hassidim Kelner Nguyen Onak][Newman Sohler][Levi Ron]
  - e.g., planar

Property testing

Main Goal:

- Quickly distinguish inputs that have specific property from those that are far from having the property



Property Testing

- Properties of any object, e.g.,
  - Functions
  - Graphs
  - Strings
  - Matrices
  - Codewords
- Model must specify
  - representation of object and allowable queries
  - notion of close/far, e.g.,
    - number of bits/words that need to be changed
    - edit distance

A simple property tester

Sortedness of a sequence

- Given: list $y_1 \, y_2 \ldots \, y_n$
- Question: is the list sorted?

- Clearly requires n steps – must look at each y_i

Sortedness of a sequence

- Given: list $y_1 y_2 \dots y_n$
- Question: can we quickly test if the list close to sorted?

What do we mean by ``quick''?

- query complexity measured in terms of list size *n*

- Our goal (if possible):
  - *Very small* compared to *n,* will go for *clog n*

What do we mean by "close"?

Definition: a list of size n is ε-close to sorted if can delete at most εn values to make it sorted. Otherwise, ε-far.

(ε is given as input, e.g., ε=1/5)

Sorted:	1	2	4	5	7	11	14	19	20	21	23	38	39	45
Close:	1	4	2	5	7	11	14	19	20	39	23	21	38	45
	1	4		5	7	11	14	19	20		23		38	45
Far:	45	39	23	1	38	4	5	21	20	19	2	7	11	14
				1		4	5					7	11	14

Requirements for algorithm:

- Pass sorted lists
- Fail lists that are ε-far.
  - Equivalently: if list likely to pass test, can change at most ε fraction of list to make it sorted

What if list not sorted, but not far?

Probability of success > ¾

(can boost it arbitrarily high by repeating several times and outputting "fail" if ever see a "fail", "pass" otherwise)

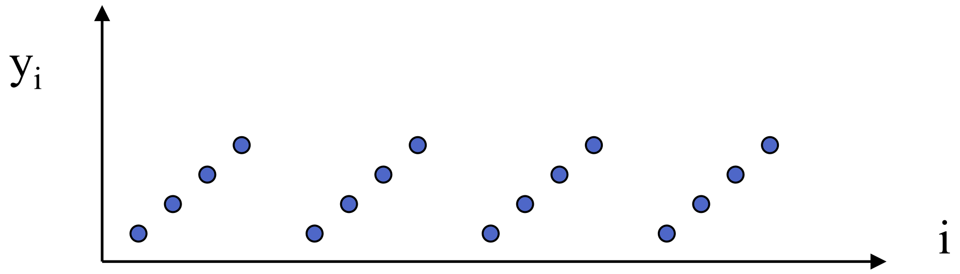
- Can test in $O(1/\varepsilon \log n)$ time

(and can't do any better!)

An attempt:

- Proposed algorithm:
  - Pick random *i* and test that $y_i \leq y_{i+1}$

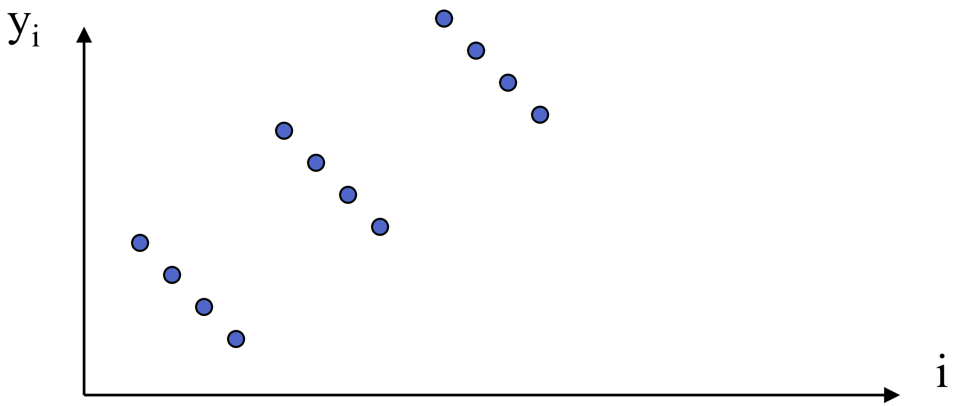
- Bad input type:
  - 1,2,3,4,5,…n/4, 1,2,….n/4, 1,2,…n/4, 1,2,…,n/4
  - Difficult for this algorithm to find "breakpoint"
  - But other tests work well…



A second attempt:

- Proposed algorithm:
  - Pick random $i<j$ and test that $y_i \le y_j$
- Bad input type:
  - $n/4$ groups of 4 decreasing elements

    4,3, 2, 1,8,7,6,5,12,11,10,9…,4k, 4k-1,4k-2,4k-3,…
  - Largest monotone sequence is n/4
  - must pick i,j in same group to see problem
  - need $\Omega(n^{1/2})$ samples



A minor simplification:

- Assume list is distinct (i.e. $x_i \neq x_j$)
- Claim: this is not really easier
  - Why?

    Can "virtually" append i to each x_i

    $x_1, x_2, \ldots x_n \rightarrow (x_1, 1), (x_2, 2), \ldots, (x_n, n)$

    e.g., 1,1,2,6,6 $\rightarrow$ *(1,1),(1,2),(2,3),(6,4),(6,5)*

    Breaks ties without changing order

A test that works

- The test:

  Test $O(1/\varepsilon)$ times:
  - Pick random i
  - Look at value of y_i
  - Do binary search for y_i
  - Does the binary search find any inconsistencies? If yes, FAIL
  - Do we end up at location i? If not FAIL

  Pass if never failed

- Running time: $O(\varepsilon^{-1} \log n)$ time
- Why does this work?

Behavior of the test:

- Define index *i* to be good if binary search for y_i successful
- $O(1/\varepsilon \log n)$ time test (restated):
  - pick $O(1/\varepsilon)$ *i*'s and pass if they are all good
- Correctness:
  - If list is sorted, then all i's good (uses distinctness) → test always passes
  - If list likely to pass test, then at least $(1-\varepsilon)n$ i's are good.
    - Main observation: good elements form increasing sequence
      - Proof: for $i<j$ both good need to show $y_i < y_j$
        - let k = least common ancestor of i,j
        - Search for i went left of k and search for j went right of k → $y_i < y_k < y_j$
    - Thus list is ε-close to monotone (delete $< \varepsilon n$ bad elements)

In closing

- These examples are just the tip of the iceberg
- Lots of cool results in the workshop this week!

Thank you