CS6820 Final Project: Sublinear Time Algorithms

Shawn Ong so396@cornell.edu
Zhen Zhang zz628@cornell.edu
Max Ruth mer335@cornell.edu

December 4, 2018

1 Introduction and Definitions

Due to the tremendous increase in available computational power in recent years, people are becoming more interested in problems that involve larger data sets. As a result, even algorithms that are linear in the input size n can be prohibitively expensive in computation time. So, the general problem is determining if a problem's solution can be found based on a small fraction of data. The conflict between a large data set and a limited amount of time gives the motivation to explore the world of algorithms which run in $o(n)$ time, or sublinear time algorithms.

For most problems, no algorithms can produce a precise solution in sublinear time. The reason is simply that the solution can be altered by changing a single element. For example, if we want to compute the arithmetic mean of a set of real numbers, it is necessary to inspect each element of the set. Indeed, this holds even for many decision problems. As a result, sublinear time algorithms often involve randomization and approximation to generate a inexact solution. However, a fast approximation is sufficient for many problems when an exact solution is computationally infeasible. For example, if we want to track properties of a large network which is rapidly changing, then by the time an exact solution is available it may no longer be valid.

In order to reason about sublinear time algorithms, we need a useful and widely applicable definition for what is a "good approximation" for a property. Formally, a *property* is a collections of inputs P; if an input $f \in P$, we say that "f satisfies property P." For instance, if we wanted to test if a graph was dense, we would consider the property P of all dense graphs. Now, consider $g : X \to Y$ as an function we wish to test, where X is a domain of dimension d representing the objects we would like to test and Y is its range. Letting $\epsilon > 0$, we say g is *ϵ-close* to satisfying a property P if there is a function $h \in P$ such that g and h are different in at most ϵd places. Otherwise, g is *ϵ-far* satisfying P. Continuing with the example of dense graphs, consider a graph with the edge characteristic function $e : [n]^2 \to \{0, 1\}$ where n is the number of vertices and $[n] = \{1, \ldots, n\}$. Then e would be ϵ-close to dense if there were only ϵn^2 edges not in the graph.

The general strategy for forming sublinear time algorithms will be to develop an algorithm called a property tester which runs in sublinear time. Loosely speaking, a property tester is a method which passes with a high probability if the property is satisfied, and fails with a high probability if the property is not satisfied. More formally, consider a property P, an input g of size d, and $\epsilon > 0$. A property tester must pass with probability of at least $2/3$ if $g \in P$ and must fail with probability of at least $2/3$ if g is ϵ-far from P. We note the value $2/3$ is arbitrary, and it could be

changed to any constant strictly greater than 1/2; performing the test repeatedly can be used to obtain an success rate arbitrary close to 1 (more importantly, the number of repetitions is sufficiently small so as not to increase the runtime asymptotically). A sublinear time algorithm to test if g satisfies P would then consist of performing the property tester $O(\log \beta^{-1})$ to say that g is ϵ-close to satisfying P with probability of error β by taking the majority answer of the property tester.

It is often the case that property testers can behave much better than is strictly required. We say a property tester has *1-sided error* if it passes with probability 1 if g satisfies property P. The speed of a property tester will be determined by the "query complexity", i.e. the number of queries of the function g the property tester performs. If the number of query complexity is independent of the input size of g, then the property is called *easily testable*. The highest standard we consider for sublinear-time algorithms within this paper consist of easily testable properties with 1-sided error.

Unless otherwise stated, all information within this paper can be found within the reference [12].

2 Examples

We highlight some interesting applications in a variety of fields of practical interest. In addition, we highlight some of the quirks of working with sublinear time algorithms, as well as common approaches to constructing them and challenges which arise due to the restrictive nature of this constraint.

2.1 Algebraic Examples

We now turn to a homomorphism testing, a problem with motivations due to program testing and probabilistically checkable proofs. Given groups $\mathcal{D}$ and $\mathcal{R}$ as well as an oracle for function $f : \mathcal{D} \to \mathcal{R}$, is it possible to tell if f is a homomorphism? It is clear that querying the oracle once for every element of $\mathcal{D}$ suffices to answer the question; it is also necessary. Indeed, given any homomorphism f, one may construct f' which differs from f at exactly one $x \in \mathcal{D}$. Instead, consider the property testing version of the same question, where, on input ϵ, the property tester should output "pass" with probabliity at least $\frac{2}{3}$ if f is a homomorphism and "fail" with probability at least $\frac{2}{3}$ if f is ϵ-far from a homomorphism, i.e. for every homomorphism $g : \mathcal{D} \to \mathcal{R}$, f and g disagree on more than $\epsilon|\mathcal{D}|$ inputs. We may try to construct such a property tester by identifying some set of relationships which must hold everywhere for a homomomorphism. Then query the oracle to see if an appropriate number of these relationships holds; if not, then we may safely output "fail." We outline this method below, restricting our attention temporarily to the case where $\mathcal{D} = \mathcal{R} = \mathbb{Z}_q$ for large $q \in \mathbb{Z}$.

It is not difficult to show that the set of homomorphisms over $\mathbb{Z}_q$ is exactly the set of functions that satisfy $\forall x \in \mathbb{Z}_q, f(x+1) - f(x) = f(1)$. So we may construct a property tester which checks whether this holds for most $x \in \mathbb{Z}_q$. However, this does not suffice, as there exist functions which are far from any homomorphism, but will pass the property tester with high probability. The example given in [12] is that of $g(x) = x \mod \lceil \sqrt{q} \rceil$, which passes the test on a $1 - \frac{1}{\sqrt{q}}$ fraction of the domain (at all x except for those which are $0 \mod \lceil \sqrt{q} \rceil$).

Instead, we may construct a property tester based on a different characterization of homomorphisms. The set of homomorphisms over group $\mathcal{D}$ can also be characterized as the set $\{f : \forall x, y \in \mathcal{D}, f(x+y) = f(x)+f(y)\}$. It then appears reasonable to test whether this relationship holds for most $x, y \in \mathcal{D}$. Indeed, one can show that, for any $\delta < \frac{2}{9}$, if $f(x+y) = f(x)+f(y)$ for at least $1-\delta$ fraction of $x, y \in \mathcal{D}$, then there exists a homomorphism g such that f is $\delta/2$-close to g.

Since this holds, we may construct a property tester based on this characterization by sampling enough pairs x, y and outputting "pass" if and only if every pair satisfies $f(x+y) = f(x)+f(y)$. All homomorphisms must always pass, and if f is ϵ-far from homomorphism, then a fraction of at least 2ϵ of all pairs $x, y \in \mathcal{D}$ must fail. Testing $O(1/\epsilon)$ provides a sufficiently high probability of failure for such f.

Both of the mentioned characterizations of homomorphisms are by local constraints, i.e. they only rely upon a small number of values of the function.

The notion that the second characterization is useful for constructing property tests while the first is not can be formalized by the notion of a *robust characterization*. A characterization is (ϵ, δ)-*robust* if any function satisfying the constraint with probability at least $1 - \delta$ is ϵ-close to some function g that satisfies all of the constraints. Informally, a characterization is robust if the "for all" quantifier can be replaced with a "for most" quantifier and still only characterize functions close to the desired set.

Application in error correcting code

Fast property test over algebraic structures has significant applications in telecommunication. Coding theory, in particular, cares about membership of vector space over finite fields and their sub-spaces. Often, it is desirable if a corrupted string can be detected fast so that the receiver can send a request for another copy. In the context of sublinear time algorithms, the question can be formulated as if there exists codes and property tester such that the property of being a codeword is easily testable. And if there are such codes, do they have good rates and error correcting properties? Before discussing which codes are locally testable (easily testable), let us define some basic concepts in coding theory.

A set of symbols Σ is called the alphabet. A string s of length n is an element of Σ^n, where $s(i)$ denotes the i^{th} character in string s. The Hamming distance l between two strings w, w' of length n is the number of indexes i such that $w(i) \neq w'(i)$. The relative distance d is the quotient of Hamming distance and string length $d = \frac{l}{n}$. The rate of a code is the ratio of the length of a message and the length of a codeword $\frac{k}{n}$. A code is (q, δ) locally testable if there exists a recursive function given random access to q queries such that codewords are mapped to 1, and strings $\delta-$far from a codeword are mapped to 0 with probability higher than $\frac{1}{2}$ [13].

The question of whether or not locally testable code exists is fairly simple. It turns out many of the well known codes are locally testable! For example, the Hardmard code, which encodes a message x from F_2^k into $F_2^{2^k}$ by taking inner product of $\langle x, y\rangle$ over all $y \in F_2^k$ is locally testable. The fact that Hardmard code is locally testable shouldn't be surprising. Let f be the encoding function, then by the linearity of inner product we have $f(x) + f(y) = f(x + y)$, which is a property of homomorphism. Moreover, Reed-Muller codes, which are widely used in wireless data transmission, as well as Reed-Solomon codes, which are used in optical discs, are all locally testable.

The question of how efficient locally testable codes can be is more difficult. There are locally testable codes "near linear" in the following sense[5]:

1. Codes with rate $n = k^{1+\epsilon}$ for arbitrary positive ϵ.
2. Codes with rate $n = k^{1+\epsilon(k)}$ where $\epsilon(k)$ is a function goes to 0 such as $\frac{1}{\log\log k}$

The current state of the art result is the construction of locally testable codes with $n = O(kpoly(logk))$. However, the problem if there exist locally testable codes with $n = O(k)$ remains open.

The last question is what are the properties of a code that makes it locally testable? Intuitively, linear codes with large relative distances between codewords and a low rate are likely to be locally testable, simply because codewords are likely to be locally distinct from other strings. Multiple studies have shown this is indeed the case[12]. Hardmard code, for example, has a rate of $\frac{k}{2^k}$ and minimal Hamming distance 2^{k-1}. The other intuitive case is when there is a strong local constraint for codewords. More specifically, codes generated by low-degree polynomials over finite fields turn out to be locally testable. These codes include Reed-Muller Code and Reed-Solomon Code.

2.2 Graph Property Testing

In this section, we will consider the testing of graph properties. In order to consider this general question, we must first define what it means for a graph G to be ϵ-far away from from satisfying a graph property P. For now, we consider dense graphs; we will say G is ϵ-far from satisfying P if there is a graph G' that satisfies P such that G and G' both have n vertices and differ by at most ϵn^2 edges. Note that, since the graph is dense, we will consider sublinear time to be $o(n^2)$.

Given a graph property, a good initial approach would be the following algorithm:

1. Choose, at random, a constant number of the vertices of G to induce a subgraph.
2. Test the induced subgraph for the property P.
3. If the subgraph has the property, pass the test. Otherwise, fail.

We will refer to an algorithm with this design as a "natural algorithm". All of the examples given later within this section are performed via natural algorithms. It is clear that if the natural algorithm acts as a property tester, then that property is easily testable.

Remarkably, graph properties which are easily tested via natural algorithms with 1-sided error can be completely characterized. Consider a graph G and property P. We say that P is *hereditary* if for any G which satisfies P, the graph induced after any vertex of G is removed also satisfies P. Furthermore, we say P is *semi-hereditary* if there exists another hereditary property H such that:

1. if G satisfies P then G also satisfies H, and
2. if $\epsilon > 0$, $n > M(\epsilon)$ for some function M, and G is ϵ-far from satisfying P, then H also is not satisfied.

In other words, this definition of semi-hereditary gives a notion of being "almost hereditary" by comparing P to another hereditary property H. With this, we may give the following theorem: P is easily testable by a natural algorithm with 1-sided error if and only if P is semi-hereditary. The proof of this theorem is given in [1], but is too involved for this project.

An example of an easily testable property of a graph is the existence of triangles. A graph G is said to be triangle-free if no induced subgraphs are isomorphic to K_3. The property tester for being triangle-free follows from the following Lemma:

Triangle Removal Lemma [12] There exists a function $q : (0,1) \to (0,1)$ such that if G is $\epsilon-$far from being triangle-free, then G contains at least $q(\epsilon)n^3$ triangles.

The protocol of the tester is the following: for any $\epsilon > 0$, sample a set $S = \frac{9}{q(\epsilon)}$ triples of vertices. Pass the test if none of them are triangles, and fail the test if any of them is a triangle. Clearly, this tester has a one-sided error because all triangle-free graphs must pass. If G is $\epsilon-$far from triangle-free, then the given lemmas shows G contains at least $q(\epsilon)n^3$ triangles. The probability of 9 triplets having no triangles is less than $\frac{1}{2}$. However, the function q is mysterious. There are upper bounds for q in the form of a tower of exponents of heights $poly(1/\epsilon)$, and obtaining better upper bounds of q is a major open problem [12].

Other graph properties that are testable in constant time (easily testable) include k-colorability and partition problems of fixed order. Just like the case of triangle-freeness, the query complexity of k-colorability in terms of ϵ remains open.

The results above apply generally to dense graphs, i.e. graphs containing $\Theta(n^2)$ edges. However, several applications involving large sparse graphs, such as social networks or transportation models, also propose interesting questions for the feasibility of sublinear time algorithms. For this section, we will be primarily interested in graphs where every vertex has degree bounded above by a fixed integer d. Whereas in the dense graph applications, we use the adjacency matrix representation, sparse graphs are more efficiently represented using adjacency list, i.e. the graph is represented by a list of neighbors for each vertex. The tester can then query the representation of G by asking for the i^{th} neighbor of any vertex v, for $1 \leq i \leq d$. Again, we must amend our working definition of ϵ-closeness to work in this framework. We say that an n-vertex graph G of degree bounded by d is *ϵ-far* from G' if at least ϵdn edges need to be added/deleted to obtain a graph isomorphic to G' (in particular, compare this to the ϵn^2 for dense graphs). Then G is ϵ-far from satisfying property $\mathcal{P}$ if G is ϵ-far from every graph of bounded degree d satisfying $\mathcal{P}$. Likewise, we will revise the definition for sublinear time to be $o(dn)$.

Unlike in dense graphs, the fact that a property is hereditary does not automatically imply that sublinear time algorithms exist for it. In particular, if a dense graph is ϵ-far from being triangle-free, then it must contain a non-k-colorable subgraph on $c(\epsilon)$ vertices (this result is due to Rödl and Duke [14]). It turns out that this is not the case in sparse graphs; even for $k = 2$, a d-regular expander of logarithmic girth is far from being bipartite, while every set of $O(\log n)$ vertices has no cycle and is therefore bipartite. In fact, for any $k \geq 3$, there are graphs which are far from being k-colorable, but all sets of $\Omega(n)$ vertices span a 3-colorable graph [4]. In this case, this requires any 1-sided error tester for 3-colorability must have query complexity $\Omega(n)$. So in bounded-degree graphs, the relation between local properties and global properties is not so strong as it is in dense graphs. Even 2-colorability requires query complexity $\Omega(\sqrt{n})$ [6]; the fact that this is greater than constant implies that it is impossible for efficient testing algorithms to exist for arbitrary hereditary properties in sparse graphs (in contrast to the result above for dense graphs).

However, that is not to say that no such sublinear time algorithm exists. Benjamini, Schramm, and Shapira [2] show that planarity can be tested with a constant number of queries, albeit with 2-sided error; moreover, their proof demonstrates that any minor-closed property (i.e. closed under the removal of edges or vertices and under contraction of edges) is testable in bounded-degree graphs. Taking the example of high-girth expanders and planarity, it is impossible to

achieve 1-sided error with $o(\log n)$ queries (such graphs are far from planar, but do not have small non-planar graphs due to high girth). The query complexity achieved by [2] is $2^{2^{2^{poly(1/\epsilon)}}}$, which was later improved to $2^{poly(1/\epsilon)}$ in [7].

2.3 Approximation Algorithms

In addition to property testing, sublinear time algorithms also lend themselves to the application of approximating classical optimization problems. As before, it is often impossible to exactly solve such problems in sublinear time; in the context of approximation algorithms, the necessary freedom is provided by allowing the algorithm to output an answer which is approximate to the optimal, i.e. within a additive or multiplicative factor of ϵ (this takes the place of the notion of ϵ-closeness).

With recent progress in technology over the past few decades, distributed algorithms have quickly gained steam as a research area. One may note the natural similarities between local distributed algorithms and sublinear time algorithms – for example, local property testing can be easily implemented by parallel computation. Indeed, Parnas and Ron [11] show that local distributed algorithms and sublinear time algorithms are related in such a way that efficient distributed algorithms for degree-bounded distributed networks can be used to construct sublinear time approximation algorithms over these same networks.

As an illustrative example, consider the vertex cover problem on graphs of degree bounded by $d \in \mathbb{N}$. Suppose that we have access to an oracle that, when queried on vertex v, indicates whether or not v is in the vertex cover. Sampling $O(1/\epsilon^2)$ queries is sufficient to get an ϵn additive approximation to the optimal vertex cover. The only issue to consider is how to obtain such an oracle; it turns out that, if there is a local distributed algorithm (i.e. it runs in time independent of the number of vertices) for vertex cover, then such an algorithm can be used to construct this oracle. For any constant k, the k-neighborhood of any vertex v, is of constant size $O(d^k)$. So we can simulate the run of any processor v in the distributed algorithm by another which is sufficiently close enough to affect v's computation (here, it is assumed that the vertex cover is computed on the graph of the distributed network, where each vertex is a processor). Parnas and Ron then use a local distributed c-approximation algorithm to get an algorithm to approximate vertex cover. The approximation algorithms constructed in this manner are neither multiplicative nor additive approximations. Instead, we say that $\hat{y}$ is an (α, β)-*approximation* to y if $y \leq \hat{y} \leq \alpha y + \beta$; this particular algorithm produces a $(c, \epsilon n)$-approximation which has query complexity $d^{O(\log(d/\epsilon^3))}$ (in particular, this is independent of n), which was later improved by Marko and Ron [8] to $d^{O(\log(d/\epsilon))}$, for a $(2, \epsilon n)$-approximation.

Other work following the same idea but constructing a different class of local distributed algorithms has led to improved approximation algorithms which incur only a polynomial dependency on d and ϵ. Nguyen and Onak [10] demonstrate an algorithmic technique that greedily and locally simulates the standard sequential greedy algorithm. Recall the standard greedy 2-approximation algorithm for vertex cover which finds an arbitrary maximal matching and returns the endpoints of these edges as a vertex cover. The matching is greedily constructed by ordering the edges, and adding each edge to the matching if neither of its endpoints are already matched. A first attempt at implementing such an algorithm in the distributed setting may be the following:

1. Randomly assign distinct ranks $r(u, v)$ to every edge (i.e. randomly fix an ordering).
2. Assign a processor to each edge (u, v).

3. Each processor considers the edges adjacent to either u or v and recursively checks whether any such edge with lower rank is already in the matching. If so, then (u, v) is not added to the matching; otherwise, it is.

As this implements the greedy algorithm, the proof of correctness follows; we need only consider the runtime. One issue that may arise is the potential for recursive calls to create chains of linear length; Nguyen and Onak show that randomly choosing r ensures that these chains are short for all processors with high probability. It follows that the expected query complexity of one oracle call can be bounded with the observation that the probability of going down a a recursion path of length k is bounded by the probability that the ranks of edges in the path are strictly increasing, which is at most $\frac{1}{k!}$. The number of vertices at distance k is at most d^k, so the numbe of explored vertices can be bounded by e^d/d, giving expected query complexity $O(e^d)$. A formal proof over several oracle calls is more involved, as different queries may share some dependencies; the details may be found in [10].

It should also be noted that this basic framework readily applies to other greedy algorithms. Nguyen and Onak use it to generate a $(1, \epsilon n)$-approximation algorithm for maximum matching with runtime $2^{O(d)}/\epsilon^2$ (independent of n), by locally constructing increasingly better matchings. Other problems for which this method produces a sublinear approximation algorithm include set cover, dominating set, and maximum weight matching [10].

3 Conclusion

From [12], it seems a large class of problems involving approximation on graph properties can be done in sublinear time. However, a problem having sublinear time approximation is a special case rather than the norm. For example, some basic problems such as approximating the cost of min-cost matching and min-cost bipartite matching are known to not have sublinear time algorithms. In fact, any constant factor approximation requires $O(n^2)$ time even if the algorithm is randomized [3].

Additionally, another computational model of algorithms in the same spirit is that of streaming algorithms. Much progress has been made recently in this field, which finds applications in IP network traffic analysis, mining text message streams, or a variety of other large data sets [9]. Broadly speaking, such algorithms concern a large amount of data such that each data point arrives quickly and can only be seen a limited number of times, and the size of persistent memory is significantly smaller than the entire dataset. Whereas sublinear time algorithms bound the runtime to be asymptotically smaller than the input, streaming algorithms do not impose such a strict runtime requirement, but rather bound memory usage as well (in addition to small per-item runtime). In a sense, this weakens the restrictions of sublinear time algorithms, as such an algorithm necessarily uses sublinear space; however, the streaming nature of the data prevents operations that may be desired in sublinear time algorithms, for example, querying the same vertex multiple times. Though it remains to be seen whether methods from the two fields lend themselves to be easily applied in the other setting, both provide methods for which modern algorithms may be applied to increasingly larger data sets.

References

[1] Noga Alon and Asaf Shapira. A Characterization of the (Natural) Graph Properties Testable with One-Sided Error. *SIAM Journal on Computing*, 37(6):1703–1727, January 2008.

[2] Itai Benjamini, Oded Schramm, and Asaf Shapira. Every Minor-closed Property of Sparse Graphs is Testable. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, pages 393–402, New York, NY, USA, 2008. ACM.

[3] Artur Czumaj and Christian Sohler. Sublinear-time algorithms. In Oded Goldreich, editor, *Property Testing*, pages 41–64. Springer-Verlag, Berlin, Heidelberg, 2010.

[4] P. Erds. On circuits and subgraphs of chromatic graphs. *Mathematika*, 9(2):170–175, December 1962.

[5] Oded Goldreich. Short locally testable codes and proofs: A survey in two parts. In Oded Goldreich, editor, *Property Testing*, pages 65–104. Springer-Verlag, Berlin, Heidelberg, 2010.

[6] Oded Goldreich and Dana Ron. A Sublinear Bipartiteness Tester for Bounded Degree Graphs. *Combinatorica*, 19(3):335–373, March 1999.

[7] A. Hassidim, J. A. Kelner, H. N. Nguyen, and K. Onak. Local graph partitions for approximation and testing. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science(FOCS)*, volume 00, pages 22–31, Oct. 2010.

[8] Sharon Marko and Dana Ron. Approximating the Distance to Properties in Bounded-degree and General Sparse Graphs. *ACM Trans. Algorithms*, 5(2):22:1–22:28, March 2009.

[9] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers Inc, 2005. Google-Books-ID: 415loiMd_c0C.

[10] Huy N. Nguyen and Krzysztof Onak. Constant-time approximation algorithms via local improvements. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 327–336. IEEE Computer Society, 2008.

[11] Michal Parnas and Dana Ron. On Approximating the Minimum Vertex Cover in Sublinear Time and the Connection to Distributed Algorithms. *Theor. Comput. Sci.*, 381:183–196, 2005.

[12] Ronitt Rubinfeld and Asaf Shapira. Sublinear Time Algorithms. *Electronic Colloquium on Computational Complexity*, 13, February 2011.

[13] Ronitt Rubinfeld and Madhu Sudan. Robust characterizations of polynomials withapplications to program testing. *SIAM J. Comput.*, 25(2):252–271, February 1996.

[14] V. Rdl and R. A. Duke. On graphs with small subgraphs of large chromatic number. *Graphs and Combinatorics*, 1(1):91–96, December 1985.