Classifying Graph Families in a Streaming Model

Amulya Musipatla and David Woodruff

***Abstract*— Given a stream of edge additions and deletions, we want to be able to classify the result graph as part of different families. This paper will outline our results with families of complete graphs as well as families with fixed vertex cover sizes.**

I. Introduction

A. Background/Problem

Our work this semester has focused on graph streaming and classifying graphs with a streaming model. Under this model, our input consists of a stream of data, where we process this data in real time. Rather than getting all of our input as a whole and being able to look at it all at once, this model constrains algorithms to take in at most a fixed amount of data at a time and choose what to do with it.

Upon receiving a new piece of data, our algorithm can make any alterations to current structures, and then we don't get to look at the data again. This successfully models dynamic networks and environments, when structures can change. This also enables algorithms to work with a lot of data while not using that much space.

We work with graph streaming, where the elements in our stream consist of edge additions and deletions. Throughout the semester, we looked at various graph families and sought to classify graphs as either being part of that family or not while using less space than one would require to store the whole graph.

B. Approach Overview

We focus on families of graphs with small vertex covers as well as families of complete graphs. In both families, we use subsampling as a main technique, by using some property of the family that would be exhibited under sampling.

In graphs with small vertex covers, we know that under any sampling, if we choose to not keep the vertices that are part of the vertex cover we would be left with an empty graph since every edge is incident to one of those vertices. We use this fact in our approach to strategically subsample and observe the effects.

In complete graphs, we take advantage of regularity properties. In a clique, we are able to remove any set of vertices and maintain a regular graph, and in any complete k-partite graph we are able to remove k vertices from different partitions to maintain regularity. This fact is exploited in our approach, and we check for regularity across many subsamplings.

C. Related Work

We try to gather information about the structure of a graph without needing to store the whole graph. Related algorithms that try to estimate connectivity involve graph sketching and creating sparsifiers. These algorithms also involve strategic subsampling to create a smaller version of the graph and successfully estimate these properties, which is what inspired us to go down a similar route. [1][2][3]

We also use an existing algorithm, the CountSketch algorithm [4], to estimate how many times some piece of data appears in a stream. We modify this so that an addition would count as seeing some edge and a deletion would have the effect of undoing that action. Using this, we would like to say that we have seen every edge 0 times in an empty graph, and we estimate the Euclidean norm of our result from the CountSketch algorithm in order to check this.

D. Contributions

Our research proposes algorithms to identify graphs with small vertex covers as well as complete graphs with a high probability. We are also able to successfully identify regular graphs, which we use as a subroutine in our classification of complete graphs. By analyzing a single stream in multiple ways at once, we are able to require only logarithmic space- which is less space than would be necessary to store the whole graph. Thus, our algorithms provide an efficient method of identifying certain families, while providing insights on the capabilities of subsampling.

E. Layout of the Paper

Section II describes our algorithm and approach to classify graphs with smaller vertex covers. This begins with a deep look into stars, followed by a way to extend this algorithm and logic to other graphs with small vertex covers.

Section III outlines our approach on complete graphs. We begin by looking at cliques, followed by specific types of complete k-partite graphs, and continuing on with our current research focusing on generalizing our algorithm to all type of complete graphs.

II. Approach on Stars and an Extension to Fixed Sized Vertex Covers

A. Main Techniques

Our proposed algorithm relies on certain properties that hold only for stars. In particular, we work with properties about disjoint edges. We say that two edges are disjoint when they don't have any endpoints in common, and we find that a star and a triangle are the only two graphs such that no two edges are disjoint. We prove this fact in the next section on evaluation of the algorithm.

The main technique of the proposed algorithm is subsampling the graph. We use a universal hash function so that we can estimate a uniform distribution while only using $\log(n)$ space to store the function. With this tool, we don't have to manually store which vertices we choose to keep and which we choose to discard. Instead, with every edge addition and deletion, we are able to dynamically figure out what we chose to do.

We also use the CountSketch algorithm to estimate the Euclidean norm of a vector. The algorithm is explained in more depth later, but we choose to use this algorithm because it uses universal hash functions and doesn't require much space.

B. Algorithm

Based on our claim that only triangles and stars maintain the property of not having any disjoint edges, our algorithm's first step is to determine that our graph is not a triangle. In order to rule out the case of a triangle, we can use sparse recovery on the stream in order to check if the number of edges is at most three, and in this case we can simply recover the graph to check if it is a triangle or a star. Since the size of the graph is fixed in this case, we only use a constant amount of space and time, and this step doesn't slow down our algorithm.

After this step, we seek to construct an algorithm that will determine if a simple graph is a star given a stream of edge additions and deletions.

Given that we can easily determine if the graph constructed by a stream is a triangle, we can now rely on the property that stars are the only graphs with no pairwise disjoint edges. We know that any non-triangle graph we look at either has at least two disjoint edges, or is a star.

We can first start out by randomly choosing to keep or discard each vertex of our graph. Then, when we see any edge (u, v) being added or deleted in our stream, we choose to do that action only if we decided to keep both u and v initially. We then want to see if the graph is empty at the end, relying on the fact that this happens more with stars than any other graph.

We can use sampling in order to check if the graph is empty given the additions and deletions we focused on. The additions and deletions would be considered as +1 and -1 respectively to a vector of edges. If the vector only contains 0s by the end, we know that the graph is empty. In order to tell if the vector is full of 0s, we can estimate the Euclidean norm, which will be 0 if and only if the vector only contains 0s.

To estimate the Euclidean norm, we can use the CountSketch algorithm, which uses a random linear map $S : \mathbb{R}^n \rightarrow \mathbb{R}^k$, a 2-wise independent (or 2-universal) hash function $h : [n] \rightarrow [k]$ and a 4-wise independent hash function from $\sigma : [n] \rightarrow \{-1, 1\}$. We can easily calculate $(Sx)_i = \sum_{j,h(j)=i} \sigma(j)x_j$ by updating the sum with each edge addition or deletion. We use $||Sx||^2$ to estimate $||x||^2$ up to a constant factor, so if our vector has all zeros, $||Sx||^2$ must always be zero. We can amplify our probability of correctly analyzing the emptiness of the graph by using this method to estimate the Euclidean norm multiple times.

Algorithm 1: Determining if stream is a star

```
1 Pick universal hash function $a : [n] \rightarrow \{0, 1\}$;
2 Pick fixed number of maps, 2-wise independent and
    4-wise independent universal hash functions (set up
    CountSketch);
3 For each addition or deletion of edge $(u, v)$ and for
    each CountSketch setup, if $a(u) = a(v) = 1$ and
    $h_i(u, v) = j$, then update each $(S_i x)_j$ by adding or
    subtracting $\sigma_i(u, v)$ to it;
4 If all $||S_i x||^2$ are 0 at the end of the stream, output
    **EMPTY;**
```

C. Evaluations

1) Complexity: We first that this algorithm should take $O(\log n)$ space, since we use a constant number of universal hash functions and a constant number of counters that each take at most $O(\log n)$ space themselves. At any point in the stream, we simply have to update points in our vector, which should take linear time. If we wanted to reevaluate with every edge addition and deletion, this would still only add a polynomial factor to our upper bound on time.

2) Disjoint Edges in a Star: As explained in the introduction, we want to find a distinct feature of a star that we can use to distinguish it from other graphs. We note that the star is one of the two types of graphs, the other being a triangle, with no two pairwise disjoint edges. That is, in a star, there are no two edges that do not share any single endpoint.

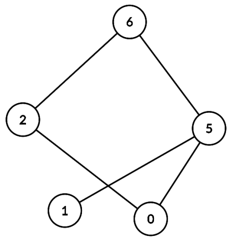

Fig. 1. In this graph, $\{2, 0\}$ and $\{2, 6\}$ wouldn't be considered disjoint, but $\{2, 0\}$ and $\{1, 5\}$ would be

We can build this argument up inductively to observe this property. To start, a graph with two edges must be a line or be disconnected, where a line can be viewed as a star. Of the graphs with three edges, the only graphs that have no disjoint edges are a triangle and a star. If we assume that our claim is true for all graphs with m edges, where $m \geq 3$, we can consider any graph with $m + 1$ edges that are not pairwise disjoint. If we remove any of its edges, all of the remaining edges must still not be disjoint since we haven't removed any vertices. This means we are left with either a triangle or a star.

In the case that we are left with a triangle, we cannot add any other edge to it without it being disjoint from an edge already in the triangle, since any additional edge will only be connected to one vertex of the triangle. If we are left with a star, then the edge we removed must have been connected to the center as well, since it would be disjoint from some other edge should we add it between two vertices or connect it to a leaf. Thus, the graph we are left with must be a star.

3) Proof of Correctness of the Algorithm: After running algorithm 1 on a graph stream, we are able to determine if the subset of edges we keep corresponds to an empty graph or if it still contains some edges. We claim that a star is more likely to be empty than any other reasonably sized graph.

Given a star graph, we know that if x is the root node and $a(x) = 0$, then the graph will necessarily be empty. This happens with probability $1/2$.

We will show that for a graph of considerable size that isn't a star, the probability of the subsampling being empty is smaller than 1/2.

To do this, we claim that the probability of surviving a sampling strictly increases as we add more edges to a graph. This is clear since with more edges, there are more possible combinations of vertices that will lead to an edge surviving.

1) *Graphs with three edges*:
   We first consider any graph with three edges. If the edges are all disjoint, then the probability of the sampling being empty is $(\frac{3}{4})^3 = \frac{27}{64} < 1/2$. If only two are connected, then the probability of the sampling being empty is $(\frac{3}{4})(\frac{5}{8}) = \frac{15}{32} < 1/2$. If they are all connected, then they either form a triangle, a star, or a line. The probability of this line or triangle not surviving a sampling is exactly 1/2. These graphs survive a subsampling with at least the same probability as a star, and will help us establish a baseline probability for larger graphs.
2) *Graphs with four edges*:
   We now look at a graph with 4 edges that is not a star. If we remove some random edge, we're left with a graph of 3 edges. If this graph is not a star, then it has probability $\geq 1/2$ of surviving the sampling from the reasoning above. Adding an edge strictly increases the probability of surviving the sampling, so our graph of 4 edges has a probability greater than 1/2 of surviving. In other words, it has a probability strictly less than 1/2 of being empty.
   If the graph left after removing an edge is a star, then the edge we removed either connects two edges or is only connected to some leaf of the star. In both cases, we see that we could instead choose a different edge to remove so that we're left with a triangle or a line, which reduces to our first case. We again have a probability strictly less than 1/2 of the sampling being empty. We see that for graphs of 4 edges, only the star has a probability greater than 1/2 of being empty after a sampling.
3) *Graphs with more than four edges*:
   We can more generally consider a graph of size $n+1 > 4$ edges. I claim that if it is not a star, it has a larger than 1/2 probability of surviving the sampling. By removing a edge, we end up with a graph that is a star or is not. If its not a star, then we know it will survive the sampling with a probability larger than 1/2, and adding the edge back will only help. If we are left with a star, then the edge we removed either connects two edges or is connected to a leaf only. In both of these cases, we can again choose to remove one of the other edges instead of the edge we originally chose as we did in the case of graphs with four edges, and we see that the claim clearly holds then.
   Thus, for all graphs of size 4 or more, we know that unless the graph is a star, the probability of the sampling being empty is less than 1/2, and we can repeat our proposed algorithm multiple times to take advantage of this and determine if our graph is a star with a higher probability.

D. Extensions

In general, when we have a family F of graphs that all have the same size vertex cover, we see that subsampling should lead to an empty graph with some fixed probability. Essentially, based on the size of the vertex cover, we can alter the probability that we keep a vertex in our sampling. Using the CountSketch algorithm described for algorithm 1, we estimate all of our Euclidean norms based on our samplings, to see when our subsampled graph ends up empty. Under different sampling rates and multiple iterations, we should be able to determine whether the theoretical probability that a graph from F survives matches the experimental results that we observe from running our algorithm on any stream. In this way, we can extend our previous algorithm to a more generalized graph with a small vertex cover.

III. Approach on Complete Graphs

A. Main Techniques

Similar to our approach on the star, we use subsampling as a main technique. We also use regularity as the key factor in determining if a graph is complete, as we see that this property is maintained more consistently for complete graphs.

B. Algorithm

The first step in our algorithm is to check for regularity. We first sample from some distribution and map each vertex in our graph to a value. We can use a universal hash function h in this step in order to minimize the amount of collisions we expect as well as the amount of space we use.

With each vertex given a random value, we maintain a sum s and proceed as follows. Whenever we see an edge (u, v) added, we add $h(u) + h(v)$ to s, and whenever we see (u, v) deleted, we subtract their hash values from s. In order to check if our graph is regular, we can keep a count of all the edges we have added to our graph, and since we know what the number of vertices is, we can determine what the degree of every vertex should be. Given this, we divide our sum s by our determined degree, and we should get a whole number if our current graph is regular. We concurrently perform this calculation with different samples from a distribution in order to be more sure of our judgment. An example of this is shown in figure 2.

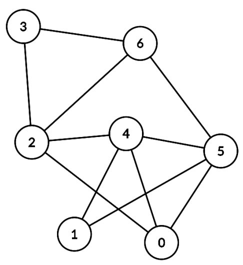
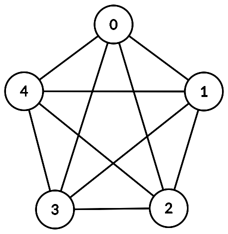

Fig. 2. If we had this graph and assigned the values depicted on each node, then our sum would be 0 + 1(2) + 2(4) + 3(2) + 4(4) + 5(4) + 6(3) = 70. We have 11 edges and 7 nodes, and since 22 isn't divisible by 7 we would be able to tell that this graph isn't regular. However, in the 5-clique, our sum is $0(4)+1(4)+2(4)+3(4)+4(4)=40$. We have 10 edges and 5 vertices, implying each vertex could have a degree of $20/5=4$, and our sum is divisible by 4.

Given that we can reliably determine a graph's regularity, we can now determine the completeness of the graphs we come across. On a complete graph, can check for regularity on the stream without any subsampling, and also check for regularity when we randomly choose one vertex to discard. Again, we can increase accuracy by repeating this analysis with different vertices removed.

Putting the above ideas together, we get Algorithm 2. We note that in the algorithm, every row of our array S represents a different case with some (or no) vertex removed, and every column corresponds to multiple checks so that we are sure of regularity.

Algorithm 2: Determining if stream is a clique

```
1 Pick fixed number $k$ of universal hash functions
    $\{h_1, ..., h_k\}$, and a random subset $\{x_1, ..., x_l\}$ of
    vertices. ;
2 Maintain a $(l+1) \times k$ array $S$ of variables, and have a
    variable $e$.;
  for each addition or deletion of $(u, v)$ do
      if add $(u, v)$ then
          $e++$ ;
          Add $h_i(u)$ to $S_{j,i}$ for every $j \in [l+1]$ and
            $i \in [k]$, as long as $u \neq x_k$;
          Add $h_i(v)$ to $S_{j,i}$ for every $j \in [l+1]$ and
            $i \in [k]$, as long as $v \neq x_k$;
      else
          $e--$ ;
          Subtract $h_i(u)$ to $S_{j,i}$ for every $j \in [l+1]$ and
            $i \in [k]$, as long as $u \neq x_k$;
          Subtract $h_i(v)$ to $S_{j,i}$ for every $j \in [l+1]$ and
            $i \in [k]$, as long as $v \neq x_k$;
      end
  end
3 If $e = \binom{n}{2}$, all sums in the first row of $S$ are divisible
    by $n-1$, and all sums in the following rows of $S$ are
    divisble by $n-2$, then output CLIQUE.;
```

C. Evaluation of Clique Algorithm

1) Complexity: We note that our algorithm has a fixed number of hash functions and a fixed number of variables. Each one takes $\log(n)$ space to keep, so our overall space compexity is $O(\log n)$. At any one step, we maintain all of these sums, and whenever we want to check for regularity we simply have to divide all of these sums by whatever we think the degree should be. Both of these actions take polynomial time with respect to the number of nodes and edges we've added, so we know that we do a polynomial amount of work at each step.

2) Proof of Correctness: First, we note that with enough iterations of our regularity check, we should be able to reliably check if a graph is indeed regular. While every sum we calculate for a regular graph should be divisible by d, if even one sum is not divisible by d, we can immediately rule out the graph as regular.

Now, in order for our algorithm to work correctly, we rely on the fact that a clique is regular whenever we remove a vertex, and that this is the only graph for which this is true.

Assume for sake of contradiction that there is a graph G that is d-regular, and that $G \setminus v$ is $(d-1)$-regular for some vertex v. When we remove v from G, we know that now the degree of each of its neighbors decreases by 1, so their degrees are all $d-1$. In order for our graph to be $(d-1)$-regular, every vertex's degree must have decreased by 1, which means that v was adjacent to every vertex in the graph. This means that there was $d+1$ vertices in total (including v), and that every vertex was connected to every other vertex in the graph. Thus, our graph G must have necessarily been a clique.

Thus, if any graph passes our regularity tests, we can reliably classify it as a complete graph.

D. Extension to Complete k-Partite Graphs

We can extend the idea of algorithm 2 to come up with a more general algorithm for a complete k-partite graph, where every part has the same size. Figure 3 illustrates a potential graph we'd want to classify. Again, we know that this graph should be regular, since each part has the same size. Rather than remove a single vertex, we will remove k vertices. If these k vertices form a clique and the rest of the vertices form a regular graph, we claim that the overall graph is complete k-partite.

In this case, it is more necessary to check for regularity given multiple sets of size k. In our more generalized algorithm, if our graph has n vertices and we are determining if it is a complete k-partite graph, we first check if the graph is $((k-1)(\frac{n}{k}))$-regular. Then, when we remove k vertices, we want to check if the graph is $((k-1)(\frac{n-k}{k}))$-regular. We illustrate two possible cases of subsampling to make this clearer in figure 4 and 5.

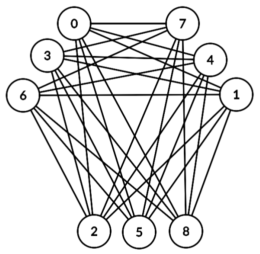

Fig. 3. A complete 3-partite graph with equal sized partitions

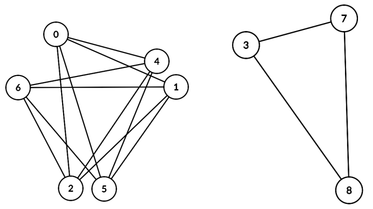

Fig. 4. A successful case of subsampling. We randomly choose 3 vertices from different partitions, and the left graph is still regular while the right graph is a clique, both of which we would be able to classify.

Algorithm 3: Determining if stream is a complete k-partite graph

```
1 Pick $l$ subsets $\{A_1, ..., A_l\}$ of $k$ vertices each. for each addition or deletion of $(u, v)$ do
      if add $(u, v)$ then
          $e++$ ;
          Maintain sums involved with the regularity check of the whole graph $G$, as well as for $G \setminus A_i$ and $A_i$ for each subset we chose.
      else
          $e--$ ;
          Maintain sums involved with the regularity check of the whole graph $G$, as well as for $G \setminus A_i$ and $A_i$ for each subset we chose.
      end
  end
2 Check that $G$ is regular and that, if $G$ induced on $A_i$ is regular, then $G \setminus A_i$ is also regular.;
```

Algorithm 3 illustrates the algorithm described at a high level. Details involving counting the number of edges, determining the degree, and other set up are left out, but are included in algorithm 2.

E. Evaluation of Complete k-Partite Algorithm

1) Complexity: Again, similar to the algorithm regarding clique classification, we only use a constant number of counters, variables, and hash functions. Each of these require $\log(n)$ space, so we use $O(\log(n))$ space in general. Given that we have a constant number of regularity checks going on, we still only require polynomial time with respect to n to maintain our sums after every edge addition and deletion. Thus, we still continue to require only polynomial time.

2) Proof of Correctness: We first look at the case with bipartite graphs. Consider any graph G with n vertices such that every vertex has degree $\frac{n}{2}$. Say we remove two vertices that have an edge between them. First, by checking if the 2 vertices form a clique, we know that if G is bipartite, then they are both in different partitions. If every other vertex's degree decreases by $(k-1) = 1$, we know that each of them could only be neighbors with one of the vertices we removed, which is true for bipartite graphs. By repeating this check with multiple pairs of vertices, we increase the probability that we actually have a bipartite graph when we pass these tests. We also note that since we keep track of how many edges we have at any one point, we know that if our graph is bipartite, then it must be complete in order to have $\frac{n^2}{4}$ edges.

Similarly, on any other graph of n vertices that we want to check is k-partite, we can check with multiple k sized subsets that our graph is still regular with the correct degree, ensuring that every remaining vertex is only connected to $k-1$ of the vertices we remove. With multiple checks of this nature, we can more accurately determine if our graph is a complete k-partite graph, also relying on the fact that we can check that every subgraph we check has the correct number of edges.

3) Possible Generalizations: We use the fact that we know the size of the graph as well as how many edges there should be in a complete k-partite graph to determine what degree every vertex should have. We could generalize this algorithm to situations when we don't know k by factoring the number of edges for our whole graph as well as the subgraphs and determining how many partitions it would require and whether it is possible for it to be a complete k-partite graph for some k.

4) Shortcomings: One of the biggest shortcomings of this classification is that it requires that all partitions of our graph have the same size. We have this requirement to ensure that our graph is regular. In cases where the partitions have different sizes, we find that only vertices in the same part would have the same degree, which means that our whole sum wouldn't always be divisible by some number k. This gives motivation for our current work.

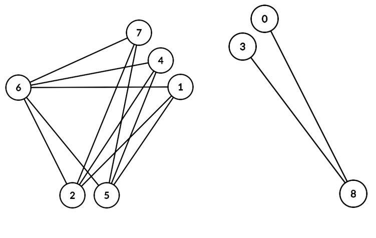

Fig. 5. A failed case of subsampling. Three vertices that aren't from different partitions are chosen, but in this case the algorithm would detect that the left graph isn't regular and that the right graph isn't a clique, so we would dismiss this as a bad sampling.

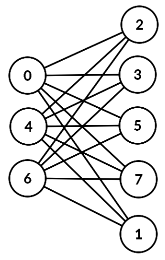

Fig. 6. An example of a potential graph we'd want to classify. Here our sum would be 5(0+4+6)+3(2+3+5+7+1) = 5(10) + 3(18)

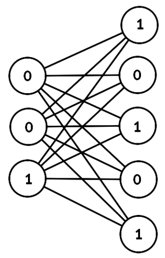

Fig. 7. An example of what subsampling vertices to contribute to our sum would look like. Here, our sum would be $5(1) + 3(3)$.

F. *Current Work with Uneven Partitions*

Currently, we've been working on extending our algorithm for classifying certain complete graphs to more general cases. The main issue is that our algorithm and regularity check isn't immediately applicable to other cases of complete k-partite graphs, when partitions don't have the same size. If we take the sum of all the edges in the same way as in our previous algorithm, we note that our sum is no longer divisible by some d, but it does maintain other properties. In particular, if our bipartite graph $G = X \cup Y$, where $|X| = x$ and $|Y| = y$, then for our sum s, we see that

$$s = ax + by,$$

where a and b are some values based on our distribution. This is illustrated in figure 6.

We see that if x and y are relatively prime, then there should always be values that satisfy that equation. Because of this, we want to be able to bound a and b, which should only be affected by our distribution, so that we can determine if our sum is actually realistic for a complete bipartite graph.

Our current work proposes using a binary distribution, where we assign every vertex to either 0 or 1 with equal probability. Essentially, our sum then only counts the vertices that we want to count. If we pick v_1 vertices from X and v_2 vertices from Y, then

$$s = v_2x + v_1y,$$

as shown in figure 7.

We expect half of the vertices on each side to be chosen to contribute to our sum, and so a and b shouldn't vary too much from $\frac{y}{2}$ and $\frac{x}{2}$, and we can place bounds that a and b should be within with high probability.

We know that we should be able to count the number of edges in our graph, and that it should be equal to xy in order for our algorithm to even consider the possibility that it is a complete bipartite graph. We've started to analyze some simple cases in which we have xy edges but we don't have a bipartite graph in order to see how the sum behaves. In particular, we've analyzed the case where most of our graph is bipartite, except we are missing some edge between X and Y, and this edge instead connects two vertices in X and Y.

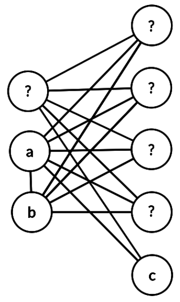

Fig. 8. An example of this edge case we consider in our analysis.

In figure 8, we see that there should be an edge between b and c, but instead there is an edge between a and c. In the cases when a and c are either both selected or both not selected to contribute to our sum, we notice that our total sum shouldn't change at all. This is because, other than these two vertices, every other vertex has the same degree that it should in a complete bipartite graph. Since a's degree is one more than it should be and c's degree is one fewer than it should be, they can potentially offset each other in half of our potential cases.

However, in the other half of the cases, when only one of them is selected, we notice that our sum changes by exactly one. In these cases, we'd ideally like any potential solutions to our equation to vary by enough for our algorithm to notice. That is, we'd like solutions to $1 = ax + by$ to require either a or b to be large enough that we would come up with an unreasonable solution to $s = v_1x + v_2y$. If we require x to be some significant factor larger than y, we see that this difference would be noticeable. However, we also note that this is a very specific case, and that our sum could change more imperceptibly in other situations.

IV. Conclusions

A. *Surprises*

In our search for easily classifiable families, we attempted to use subsampling as a technique in many different families. One of the surprising families that we looked at briefly was the family of cycles, which we found difficult to classify. Unlike regular graphs and graphs with small vertex covers, cycles don't seem to have a distinctive property that holds under sampling. If we were to randomly sample any cycle, we have no way of being able to piece together whether it was a cycle originally, since it could mimic many types of graph under sampling.

While we expected cycles to be an easier family to classify because of it is a natural family, we learned that subsampling is only a viable technique when we are able to strategically pick out vertices or edges to keep, and when we can count on some property holding under sampling. This helped us when we focused on complete graphs, and realized that regularity was a property that would hold.

We also initially began this project as a look into directed trees. Originally, we had wanted to be able to identify the root node of a directed tree using minimal space and time. In the process of approaching this problem, we took a look at stars as an easy case. We thought that classifying stars would give us more insight into what makes a tree easier to work with, and instead we realized that this was an unexplored problem that would be useful to solve other problems.

B. Implications and Future Work

Our research exhibits a way to be able to classify graphs without needing all pieces of information. Instead of polynomial space with respect to our graph, both of our algorithms only require logarithmic space while not adding too much time complexity overhead or compromising accuracy. In all, these algorithms lay solid foundations for future work in classifying graph families through the insights we gained with subsampling and other techniques.

Being able to classify certain types of graphs could also aid in many other algorithms. Our research and possible extensions allow identification of certain structures a graph could possibly have, which could prove useful in approaching other problems.

In the future, we plan to continue to look at the last case discussed in section III, regarding partitions of different sizes. Coming up with a successful approach on these kinds of cases will provide us with more insight, and allow us to classify graphs as any type of complete graph reliably.

If one were to expand on our research, it would be beneficial to actually attempt to implement these algorithms. In theory, we expect that these algorithms should be both accurate and should use a minimal amount of space. Actually experimenting with code and analyzing results would give more insight on if subsampling is actually a realistic approach, and if our regularity check performs as well as expected. Based on the data from this work, we would be able to alter our approaches and see which parts of the algorithm take up more time/space than expected. In all, this would allow fine tuning of our approach. A more distinct extension of our research could involve expanding the techniques used to other graph families of particular interest.

References

[1] Ahn, Kook Jin, et al. Analyzing Graph Structure via Linear Measurements. *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, 2012, pp. 459467., doi:10.1137/1.9781611973099.40.

[2] Feigenbaum, Joan, et al. Graph Distances in the Data-Stream Model. *SIAM Journal on Computing*, vol. 38, no. 5, 2009, pp. 17091727., doi:10.1137/070683155.

[3] Mcgregor, Andrew. Graph Stream Algorithms. *ACM SIGMOD Record* vol. 43, no. 1, 2014, pp. 920., doi:10.1145/2627692.2627694.

[4] M. Charikar, K. Chen, and M. Farach-Colton, Finding frequent items in data streams, *Proc. ICALP*, pp. 693703, 2002.