CS5234: Combinatorial and Graph Algorithms Lecture 2,3

Graph Sketches

Lecturer: Seth Gilbert *August 18-25, 2016*

Abstract

Today, we will focus on "sketching" a graph using the L0-samplers that we have previously developed. We will then use the graph sketches to solve a variety of graph problems, including finding connected components, testing k-connectivity, and finding an (approximate) minimum spanning tree.

1 Graph Connectivity

Our primary goal is to decide whether a graph $G = (V, E)$ is connected. We will also be able to answer questions like, "how many connected components does G have?" and "are nodes u and v connected?"

We begin with a very simple algorithm that assumes we have complete access to graph G. We could, of course, simply use DFS or BFS to solve the problem in this case. We instead focus on an alternative approach that will be easier to extend to the streaming model. (This approach is essentially a simplified version of Boruvka's Algorithm for finding a minimum spanning tree.) The basic algorithm is as follows:

Repeat until there are no edges left in the graph:

- Choose a node u.
- Choose an edge $e = (u, v)$ adjacent to u.
- Merge nodes u and v.

At each step, we essentially are choosing an edge (u, v) and deleting it, while merging u and v. After $n - 1$ such merge steps, we are done: there are no edges left in the graph, and each remaining node represents a single connected component.

For today's purposes, we are not going to worry too much about the cost of implementing this algorithm. It can, in fact, be implemented in $O(m \log n)$ time. Below, we give a brief summary.

Analysis. If we want to analyze the performance of this algorithm, we need to specify more carefully how nodes are merged. To merge u and v, we create a new node u' to replace u and v, and we add an edge adjacent to u' for every edge that is adjacent to u or adjacent to v. If you think of the graph stored as an adjacency list, this means that we concatenate the adjacency lists of u and v to create the adjacency list for u'; then we remove duplicates and update the other nodes in the graph. If u and v have d neighbors in total, then this will take $O(d)$ time. In every iteration of the loop, two nodes are merged, and hence the total running time is $O(\Delta n)$, where Δ is the maximum degree of the graph.

If we want to be a bit more efficient, we can mark and compress many edges at once (instead of one edge at a time):

Repeat until there are no edges left in the graph:

For every node u in the graph:

* Choose an edge $e = (u, v)$ adjacent to u.
* Mark edge e.

Enumerate all the edges in the graph, merging all the marked edges.

Notice that the merge step can now be implemented somewhat more efficiently: first, do a DFS in the graph, and label each of the connected components induced by the marked edges. (That is, if two nodes are connected by a path of marked edges, they are in the same connected component.) For every such connected component, create a node in a new graph. Now, enumerate every edge $e = (x, y)$ in the original graph and if the edge connects two different connected components, then add an edge to the new graph. Notice that each of these merge steps can be implemented in $O(m)$ time. And each merge step at least reduces the number of nodes by 2, hence there are at most $O(\log n)$ such merge steps. Hence the total running time is $O(m \log n)$.

2 Streaming a Graph

We now consider the model where the graph is given as a stream of updates:

$$S = \langle (e_1, `add'), (e_2, `add'), (e_3, `delete'), (e_4, `add'), \ldots \rangle .$$

We will assume that the graph is initially, and each 'add' operation creates a new edge and each 'delete' operation removes an existing edge. (We will not allow a sequence of operations like $\langle ((u, v), `add'), ((u, v), `add') \rangle$ where the same edge is added twice without being deleted first.)

Assume that we know that the graph has n nodes, and that the nodes have identifiers $\{1, \ldots, n\}$. (This assumption simply makes the discussion easier, but is not particularly important.)

Our goal is to store enough information about each node so that we can run the connectivity algorithm described above.

2.1 Simple Solution

A simple solution is to store, for each node v, all the edges adjacent to v. Let $e(v)$ be a vector of dimension $\binom{n}{2}$. We want one component of vector $e(v)$ for every possible edge in the graph. We will index the vector by edge names, i.e., we will refer to the components of $e(v)$ by two indices: $e(v)_{i,j}$ refers to edge (i, j). We only want to refer to each edge once, i.e., edge $(i, j) = (j, i)$ since the graph is undirected. Thus we will always index the vector $e(v)_{i,j}$ where $i < j$. Note that the vector $e(v)$ has $\binom{n}{2}$ indices, i.e., is of dimension $\binom{n}{2}$.

For every edge (u, v) that is added to the graph in the stream, where $u < v$, we will set $e(u)_{u,v} = 1$ and $e(v)_{u,v} = 1$. For every edge (u, v) that is deleted from the graph in the stream, where $u < v$, we will set $e(u)_{u,v} = 0$ and $e(v)_{u,v} = 0$. Thus, at any given time, the vector $e(u)$ represents exactly the edges adjacent to u.

This certainly provides us enough information to run the connectivity algorithm above. Recall that in order to find the connectivity of G: (i) we choose a node u, (ii) find an adjacent edge $e = (u, v)$, and (iii) merge u and v. To implement that here, we repeatedly:

- Find a node u where $e(u)$ is not identically zero. This ensures that u has at least one adjacent edge.
- Find a node v where $e(u)_{u,v} = 1$. To implement this, we can simply examine the vector $e(u)$ and find a non-zero entry in the vector.
- Merge u and v. We can do this by deleting node v, adding every edge adjacent to v to the vector $e(u)$, and updating all the neighbors of v to have an edge to u instead of v.

After repeating these three steps at most $n - 1$ times, there are no more edges left in the graph. At this point, the number of remaining nodes represents the number of connected components.

The algorithm clearly works correctly (as it implements the scheme above). However, there are two problems. First, it obviously uses too much space, as it stores a vector of size $\Theta(n^2)$ for every node in the graph! The total space is $\Theta(n^3)$. We will use sketches to avoid this. Second, merging nodes is slow and complicated.

2.2 A Better Vector Representation

We would like an efficient way to merge vectors simply by adding them. Ideally, given two vector $e(u)$ and $e(v)$, we could calculate the merged version as $e(u) + e(v)$. Let's try something a little more clever. For each (u, v) where $u < v$, set:

- $e(u)_{u,v} = 1$
- $e(v)_{u,v} = -1$

That is, the smaller node sets the edge equal to 1, and the larger node sets the edge equal to -1. Either 1 or -1 indicate that there is an edge, i.e., every non-zero component represents an edge. As before, as the stream is presented to us, we store the vectors $e(u)$ for all nodes u, and then we use these vectors to calculate the connected components.

Now, think about what happens when we add $e(u) + e(v)$, if there is an edge (u, v). Define $e(u + v) = e(u) + e(v)$. In this case, $1 + -1 = 0$, i.e., the values cancel out, and we are left with $e(u + v)_{u,v} = 0$. Notice that every non-zero component of $e(u + v)$ represents an edge adjacent to u or v (while the component representing the edge between u and v is zero).

We will now think of sets of nodes. Let $C \subseteq V$ be a set of nodes in the graph. We can now define vector $e(C) = \sum_{u \in C} e(u)$. Notice that if, for some edge u, v where $u < v$:

- $e(C)_{u,v} = 1$, then there is an edge from a node $u \in S$ to a node $v \notin S$.
- $e(C)_{u,v} = -1$, then there is an edge from a node $v \in S$ to a node $u \notin S$.
- $e(C)_{u,v} = 0$, then either there is no edge (u, v) in the graph, or both $u, v \in S$, or both $u, v \notin S$.

This then yields the following algorithm for finding the connected components for the graph G. We will initially store each node in a set C_u. (Each such set initially holds only one node.) Associated with each set is a vector $e(C_u)$, initially equal to $e(u)$. As the algorithm proceeds, it will merge the sets together until there remains only one per connected component.

Repeat $\log n$ times:

For each remaining set C:

* Use $e(C)$ to identify an edge (u, v) where $e(C)_{u,v} \neq 0$. If no such edge exists, then skip the remaining steps.
* Assume, w.l.o.g., that $u \in C$.
* Let C' be the set containing v.
* Merge sets C and C': let $C = C \cup C'$.
* Merge the vector: $e(C) = e(C) + e(C')$.

Initially, we have n sets, one per node. In each iteration of the 'for' loop, we iterate through all the sets, merging them with another set. Afer the first iteration, if the graph is initially connected, then at most $n/2$ sets remain. After $\log n$ iterations, if the graph is initially connected, then only one set remains.

If the graph is not initially connected, then in each iteration we half the number of sets representing a connected component. By the end of the algorithm, each connected component is represented by one set.

Notice that the additive property of the vectors makes it very easy to handle the merging process: simply merge the two sets and add the two vectors. The key remaining problem is that the vectors are too large: we do not want to store the entire vector.

2.3 Graph Sketches

Instead of storing the vector $e(u)$ or the vector $e(C)$, we are going to store a sketch of the vector. Let $S(u)$ be an L0-sampler for the vector $e(u)$. Whenever the stream contains an instruction to add edge (u, v), where $u < v$, we:

- give $((u, v), +1)$ to the sampler $S(u)$,
- give $((u, v), -1)$ to the sampler $S(v)$

Whenever the stream contains an instruction to delete edge (u, v), where $u < v$, we:

- give $((u, v), -1)$ to the sampler $S(u)$,
- give $((u, v), +1)$ to the sampler $S(v)$

Notice that the stream given to $S(u)$ represents exactly the vector $e(u)$, and the stream given to $S(v)$ represents exactly the vector $e(v)$.

Now, when the stream is complete, we have a set of samplers $\{S(1), \ldots, S(n)\}$ to run the algorithm above. As before, we begin by creating a collection of sets, one for each node in the graph: $\{C_u : u \in V\}$. Each set contains exactly one node. Now, instead of storing the vector $e(C_u)$, instead we associate the sampler $S(u)$ with the set C_u. We then proceed to run the merging algorithm as above:

Repeat $\log n$ times:

For each remaining set C:

* Use $S(C)$ to identify an edge (u, v) where $e(C)_{u,v} \neq 0$. If no such edge exists, then skip the remaining steps and move set C to a collection of completed sets.
* Assume, w.l.o.g., that $u \in C$.
* Let C' be the set containing v.
* Merge sets C and C': let $C = C \cup C'$.
* Compute the new merged sketch: $S(C) = S(C) + S(C')$. Note that $S(C)$ now represents the vector $e(C) + e(C')$ that we are not storing.

Notice that the algorithm proceeds exactly as before. There are only two differences. When we need to find an edge adjacent to a set C, we use the sampler $S(C)$ associated with that set, instead of examining the vector $e(C)$ directly. The second difference is that instead of adding the vectors, we add the samplers, taking advantage of the fact that they are linear sketches. (Recall that we can add the sketches created from the L0-samplers because they consist of a set of components that can be linearly added.)

Independence. There remains one problem: independence. Notice that, as written above, we use each sketch several times, as we merge it over and over again. These uses are not independent.

To solve this problem, we implement each sampler $S(C)$ as a sequence of $\log n$ base samplers:

$$S(C)_1, S(C)_2, \ldots, S(C)_{\log n}.$$

When we merge two samplers, we simply add all $\log n$ of the component samplers.

Now in iteration i of the main loop, when we need to identify an edge (u, v) where $e(C)_{u,v} \neq 0$, we use the sampler $S(C)_i$. That is, in each iteration, we use a different base sampler. Since there are at most $\log n$ iterations, this only increases are overhead by a $\log n$ factor.

Implementation details. There are a few further implementation details. The algorithm begins with each node in a set and merges the sets as the algorithm proceeds. In order to implement this efficiently, the sets should be implemented by some sort of mergable data structure. For example, each set might be implemented using a (2,3,4)-tree (where the nodes are ordered arbitrarily, not in order by id). A $(2, 3, 4)$-tree supports a merge operation wherein two trees T_1 and T_2 can be merged as long as every item in T_1 is less than every item in T_2. If we keep the sets approximately the same size, then merge operations will cost $O(1)$. (See the exercise on this week's problem set.) For a given set C, let $T(C)$ be the $(2, 3, 4)$-tree representing set C.

A second implementation detail is that we need to find the set containing a given node. For example, imagine that we have a set C and we query the sampler $S(C)$ which returns an edge (u, v) where $u \in C$. We need to find the set containing node v. Here, the easiest solution is to assume we have a dictionary which points to the node in the $(2, 3, 4)$-tree that contains that node. For example, imagine we have another $(2, 3, 4)$-tree D which contains all the nodes in V stored in-order so that we can search for a given node. (Notice this is different from the sets in which the nodes are not stored in-order.) Each node in v in the tree D contains a pointer to the node in the *other* $(2, 3, 4)$-tree that represents the set in which the element is currently stored. For example, if u is in the set C, it is stored in the $(2, 3, 4)$-tree $T(C)$ and in the dictionary D; there is a pointer from the node representing u in D to the node representing u in $T(C)$, and vice-versa. Whenever we modify the node representing u in $T(C)$, we update the pointer properly in D.

Therefore, we can lookup a node v in $O(\log n)$ time in D and find the tree $T(C)$ in which it belongs. (Once we have found v in $T(C)$, we can walk up the tree to the root to identify the set C.) We can merge sets in $O(\log n)$ time (or $O(1)$ time if we keep the sets approximately the same size.)

Error analysis. The final algorithm contains $n \log n$ L0-samplers: $\log n$ samplers for each of the n nodes. We know that each sampler is correct with probability at least $1 - \epsilon$. Therefore, we choose $\epsilon = 1/n^{2+c}$ for some constant c. By a union bound, we conclude that every sampler is correct with probability at least $1 - (n \log n)/n^{2+c} \geq 1 - 1/n^c$.

Running time. For each element in the stream, we need to process the item by the appropriate samplers. For a given edge (u, v), there are $2 \log n$ L0-samplers to update, i.e., the samplers for node u and the samplers for node v.

For each $L0$-sampler, there are $\log n$ s-sparse samplers that might be updated. In fact, in expectation, only $O(1)$ of the $L0$-samplers need to be updated since the probabilities decrease geometrically: the first sampler include the specified edge with probability $1/2$, the next $1/4$, the next $1/8$, and so on. Hence the expected number of s-sparse samplers is $O(1)$.

Each s-sparse sampler involves $O(\log(n/\epsilon)) = O(\log n)$ hash function calculations, each of which leads to updating a 1-sampler. And each 1-sampler results in $O(1)$ updates.

Thus, in total, the cost of each edge update in the stream is $O(\log^2 n)$ in expectation, and $O(\log^3 n)$ in the worst-case.

Next, we look at the running time of the connectivity reconstruction algorithm. Overall, the outer loop runs $O(\log n)$ times. Assume that at the beginning of the outer loop, there are k sets remaining that have at least one outgoing edge.

For each of these k sets, we need to query one $L0$-sampler. (Even though each set has $\log n$ L0-samplers associated with it, there is only one designated sampler to use for each iteration of the outer loop.) To query an $L0$-sampler, we need to query up to $O(\log n)$ s-samplers. To query an s-sampler, we need to examine $O(\log n)$ 1-samplers. And each of the 1-samplers costs $O(1)$ to query. Hence the total cost of querying the sampler is $O(\log^2 n)$.

Once we have found an edge (u, v), we need to find the set containing the other edge, which costs at most $O(\log n)$. We then need to merge the two sets, which costs at most $O(\log n)$.

We also need to combine the two sketches. This requires combining two sets of $\log n$ L0-samplers. Two add two L0-samplers requires adding $\log n$ s-samplers. To combine two s-samplers requires combining two sets of $s \log n = O(\log^2 n)$ 1-samplers. And merging two 1-samplers takes $O(1)$ time. Hence the total cost for merging the two sketches is $O(\log^4 n)$. Therefore, the total cost of one iteration of the outer loop, where we begin with at most k incomplete sets, is $O(k \log^4 n)$. Since we half the number of incomplete sets in each iteration of the outer loop, we

get the recurrence relation for the total time as: $T(n) \leq T(n/2) + O(n \log^4 n)$ which equals $O(n \log^4 n)$.

Notice that the cost of merging two sketches is the most expensive part of each step. By using sketches that are faster to merge, we can reduce the log-factors on the reconstruction algorithm.

Space analysis. Finally, let us look at the total space used by this algorithm. For each of the n nodes in the graph, we store $\log n$ L0-samplers. Each $L0$-sampler requires $O(\log^4(n/\epsilon)) = O(\log^4 n)$ space. Thus the total space usage is $O(n \log^5 n)$.

In conclusion, we have shown the following:

Theorem 1 *For any constant $c \geq 1$, we have given an algorithm for determining whether a stream S represents a connected graph. The algorithm returns the correct answer with probability at least $1 - 1/n^c$. The algorithm uses $O(n \log^5 n)$ space. Processing each element in the stream takes $O(\log^3 n)$, in the worst-case, and determining whether the graph is connected takes time $O(n \log^5 n)$.*

3 k-Connectivity

We have already shown that we can determine whether a stream S represents a connected graph. A natural generalization is the question of whether a graph is k-connected. Recall:

Definition 2 *A graph is k-connected if for every cut in the graph $(S, V \setminus S)$, there are at least k edges crossing the cut.*

This is equivalent to saying that for every pair of nodes u and v, there are two edge-disjoint paths connecting u and v.

For now, we will focus on finding an algorithm for deciding if a stream S represents a 2-connected graph. You can generalize it to k-connectivity.

3.1 Non-streaming Algorithm

Consider the following algorithm for deciding if a graph $G = (V, E)$ is 2-connected:

1. Find a spanning forest T_1 of G. If T_1 does not span all the nodes in V, return false.
2. Remove the edge in T_1 from G, i.e., $E = E \setminus edges(T_1)$.
3. Find a spanning forest T_2 of G.
4. Check if $T_1 \cup T_2$ is 2-connected. If not, return false.
5. Return true.

We can decide if $T_1 \cup T_2$ is 2-connected using standard techniques. For example, we can take all pairs of nodes (u, v) and find the maximum flow between u and v, assuming each edge has capacity 1; if the maximum flow for each pair is at least 2, then we conclude that the graph is 2-connected.

Why does this work? Imagine that we have found spanning forests T_1 and T_2 (regardless of whether they are connected). There are two key claims:

Claim 3 *If $T_1 \cup T_2$ is 2-connected, then G is 2-connected.*

Proof This claim follows immediately: fix two nodes u and v, let paths P_1 and P_2 be the two edge-disjoint paths between u and v in $T_1 \cup T_2$. (We know this exists, because $T_1 \cup T_2$ is 2-connected.) By construction, P_1 and P_2 are edge disjoint and all the edges in P_1 and P_2 can be found in G. Hence there are at least 2 edge-disjoint paths between u and v in G. Since this holds for all pairs of nodes u, v, we conclude the graph is 2-connected. □

Claim 4 *If G is 2-connected, then $T_1 \cup T_2$ is 2-connected.*

Proof Let $(S, V \setminus S)$ be a cut in G. Let (u, v) and (w, z) be two edges across the cut in G. (Since G is 2-connected, we know there are at least two edges across the cut.) Are these two edges in $T_1 \cup T_2$? If both of the edges are in $T_1 \cup T_2$, then we are done: we have identified two edges across the cut in $T_1 \cup T_2$.

Otherwise, assume (w.l.o.g.) that edge (u, v) crosses the cut $(S, V \setminus S)$ in G, but is in neither tree T_1 nor tree T_2. If u and v were not connected in T_1, then we would have added edge (u, v) to T_1 since T_1 is a spanning forest. Similarly, if u and v were not connected in T_2, and edge (u, v) is not in tree T_1, then we would have added the edge to T_2, since tree T_2 is a spanning forest of $G \setminus T_1$. So in this case, we conclude that u and v must be connected in both T_1 and T_2.

Let P_1 be the path from u to v in T_1 and P_2 be the path from u to v in T_2. These are edge disjoint paths, and hence there are at least two edges cross the cut $(S, V \setminus S)$ in $T_1 \cup T_2$.

Since this is true for all cuts, we conclude that $T_1 \cup T_2$ is 2-connected. □

We conclude that $T_1 \cup T_2$ is 2-connected if-and-only-if G is 2-connected, and so the algorithm is correct.

3.2 Streaming Algorithm

Now, we can implement this algorithm using sketches. As we observe a stream S, we store 2 sketches S_1 and S_2. Each of these sketches is sufficient to reconstruct a spanning tree of the graph, as discussed above.

When the stream is complete, we first use sketch S_1 to construct a spanning tree T_1 of the graph. That is, we repeatedly use the L0-samplers to find edges connecting distinct components of the graph until we have a spanning tree; as each edge is discovered, we add it to tree T_1. Now, we check whether T_1 spans the entire graph. If not, the graph is not connected and so we return false.

Next, we need to update sketch S_2 before we use it. For each edge $e \in T_1$ we delete edge e from S_2. Remember, S_2 is just a sketch of some graph and we can add or delete edges from the graph represented by S_2. Luckily, the edges selected by S_1 are completely independent of the edges included in S_2.

Once we have deleted all the edges in T_1 from S_2, we then proceed to find a spanning forest T_2. We then take all the edges in $T_1 \cup T_2$ and run a classical algorithm to decide whether $T_1 \cup T_2$ is 2-connected.

As long as both of the two sketches work correctly, the resulting algorithm will return the correct answer. Thus, the algorithm will return the correct answer with probability at least $1 - 2/n^c$, while using two sketches worth of space.

4 Minimum Spanning Tree

Assume you have a weighted graph $G = (V, E)$ where each edge e has an integer weight $w(e) \in [1, W]$, i.e., between 1 and W. In this case, we can again use graph sketches to develop an algorithm for finding a minimum spanning tree.

4.1 Exact MST

Consider the following non-streaming algorithm for finding a minimum spanning tree, based on Kruskal's Algorithm. Let G_i be the graph containing all the edges of weight $= i$.

- Let T be an empty tree.
- For $i = 1$ to W do:
  - For every edge $e = (u, v) \in T$, merge nodes (u, v) in G_i.
  - Find a spanning forest of G_i, and add every edge to T. That if, repeatedly sample edges in G_i, merging components and adding edges to T until there are no more edges in G_i.

Notice that this is really just an implementation of Kruskal's Algorithm, where we consider edges in order of weight. For each edge in a graph G_i, if it connects two nodes that have already been connected by smaller weight edges, then the nodes will have already been merged. Otherwise, if the edge connects two previously disjoint components, then it gets added.

Again, we can clearly implement this algorithm using sketches. We maintain W sketches $S_1, \ldots, S_W$, one for each graph G_i. Whenever we see an edge of weight i, we process it using sketch S_i. When the algorithm is complete, we examine each of the sketches in order, starting with S_1. We begin with an empty tree T.

For a sketch S_i, we first merge all the nodes (u, v) where (u, v) is an edge in T. We then build a spanning forest T_i of the remaining nodes in G_i, using the algorithm in the previous section. Finally, we add all the edges from T_i to T. We continue this until T is a spanning tree of G, or until we have processed all the sketches.

Notice that this algorithm uses $O(Wn \log^5 n)$ space, and calculates a minimum spanning tree with probability $1 - W/n^c$. By choosing c sufficiently large, we can reduce this probability of error to something reasonably small. (As long W is polynomial in n, this will work.)

4.2 Approximate MST

It turns out we can reduce the space and time of the algorithm if we are willing to calculate an approximate minimum spanning tree, instead of an exact minimum spanning tree. Let T_{mst} be the real minimum spanning tree of a graph. Our goal will be to find a spanning tree T such that:

$$|T_{mst}| \leq |T| \leq (1 + \epsilon)T_{mst}$$

That is, the weight of the tree T will be at most $(1 + \epsilon)$ times the weight of the minimum spanning tree. For today's purposes, we will fix $\epsilon = 1$, and find a 2-approximation. This can be readily generalized to arbitrary ϵ.

Assume we are given a graph G. Construct a new graph G' where, for every edge e in the graph G, we round up the weight of e in G' to the nearest power of 2. That is, set $w(e) = 2^{\lceil \log w(e) \rceil}$.

Claim 5 *If T is the minimum spanning tree of G and T' is the minimum spanning tree of G', then $|T| \leq |T'| \leq 2|T|$.*

Proof Clearly the minimum spanning tree of G' is at least as large as the minimum spanning tree of G—if not, we could take the same edges in T' and we would find a cheaper minimum spanning tree of G. Similarly, we can take the spanning tree defined by the edges in T in G'. Each of these edges increased by at most a factor of 2 in weight, and so this tree has a weight of at most $2|T|$. Since it is also a spanning tree of G', we conclude that $|T'| \leq 2|T|$. □

Now, we run the algorithm from the previous section for finding an exact minimum spanning tree in G'. Notice, however, that the graph G' only has $\log W$ different weights, i.e., we only need $\log W$ different graph sketches:

$S_1, S_2, S_4, S_8, \ldots, S_W$. Similarly, when we compute the result, we only need to construct $\log W$ different spanning forests T_i.

Thus, we get an algorithm that uses only $\log W n \log^5 n$ space, and is correct with probability at least $1 - \log W/n^c$.

If you want to generalize this for arbitrary approximation factors $(1+\epsilon)$, then instead of rounding the weight of each edge e to the nearest power of 2, instead round the weight of each edge to the nearest power of $(1+\epsilon)$. You will get a better approximation of the minimum spanning tree, and the space usage will increase accordingly.

5 Counting Triangles

Assume we are given a graph $G = (V, E)$. Our goal is to find the number of triangles in G, i.e., the number of triplets of nodes (u, v, w) where $(u, v), (v, w), (wu) \in E$. Unfortunately, there are lower bounds showing that in general this can require $\Omega(n^2)$ space, i.e., we might as well store the entire graph.

If the graph has many triangles, however, we can do better. We are going to assume for now that the graph has at least n^2 triangles. (Notice that there are $\binom{n}{3} \approx n^3$ possible triangles in total.)

We are also going to assume, for now, that we can use the $L0$-sampler to get an estimate of the *number* of non-zero indices, as well as sampling a random index. We can later investigate how to approximate $||v||_0$, i.e., to find the approximate number of non-zero indices.

5.1 Setting up the sampler

Our first step is to represent the problem in terms of a vector that we can easily update as edges are added and removed from the graph. In this case, we will create a vector v of dimension $\binom{n}{3}$, i.e., with one index for every triplet of nodes—that is, for every possible triangle. We will refer to $v[i, j, k]$ as the entry in the vector associated with the nodes (i, j, k). (As before, we will assume that these nodes are in order, i.e., $i < j < k$.)

The value of $v[i, j, k]$ should equal the number of edges connecting these three nodes. For example, if i, j, k is a triangle, the $v[i, j, k] = 3$. If, instead, there are no edges connected i, j, k, then $v[i, j, k] = 0$. Or, if there is one edge (i, j), but no edges (i, k) or (j, k) then $v[i, j, k] = 1$.

Whenever a new edge (u, v) arrives, we need to update the vector v. The edge (u, v) is part of at most $n - 2$ other triangles, and so there are $n - 2$ positions in the vector that need to be updated by adding 1. Similarly, when edge (u, v) is deleted, there are $n - 2$ positions in the vector that need to be updated by subtracting 1.

As before, we can of course store this vector v in a sketch S.

5.2 Sampling

Assume that we are given a sketch S and we use the L0-sampler to identify a random non-zero index. The sketch can also return the value of v at that index. There are three possibilities: the vector is either 1, 2, or 3. Let T_3 be the number of triangles in G. Let T_0 be the number of non-zero indices in G.

We conclude that the sampler returns a triangle in G with probability T_3/T_0. Or, to put it differently, let x be a random variable where $x = 1$ if the sampler returns an index containing a 3, and zero otherwise. Then, $\mathrm{E}[x] = T_3/T_0$.

Assume, for now, that we can estimate T_0, e.g., we have some value $t = (T_0 \pm \epsilon)$. Then $t\mathrm{E}[x]$ is going to be a good estimate of T_3.

Instead of sampling once, assume we sample s times from s different sketches of the vector v, i.e., $S_1, S_2, \ldots, S_s$. (We need s different sketches of the same vector to ensure independence.) Let $x_1, x_2, \ldots, x_s$ be the random variables associated with each sample, where $x_j = 1$ if sampler S_j returns an index that is equal to 3; otherwise, let $x_j = 0$.

Notice that $\mathrm{E}[x_j] = T_3/T_0$. And if $X = \sum(x_j)$, then $\mathrm{E}[X] = sT_3/T_0$. We can now apply a Chernoff bound and show that $\Pr[|X - \mathrm{E}[X]| \geq \delta \mathrm{E}[X]] \leq 2e^{-sT_3\delta^2/(3T_0)}$.

In this case, we choose $s = 12n/\delta^2$. We also remember that $T_3 \geq n^2$, and $T_0 \leq n^3$. Thus, we conclude that the probability of error is $\leq 2e^{-4} \leq 1/8$. (Notice that we needed the assumption that there were a lot of triangles to ensure that this probability of error were small. If there were very few triangles to find, we would need a lot more samples to find them!)

What have we shown? That $X - sT_3/T_0 \leq \delta(sT_3/T_0)$, i.e., $X \leq (1+\delta)(sT_3/T_0)$. Similarly, $X \geq (1+\delta)(sT_3/T_0$. Thus, we let $Y = X/s = (1 \pm \delta)T_3/T_0$. That is, Y is a good estimate of T_3/T_0.

Now, assume that we have an estimate t such that $T_0/c \leq cT_0$, i.e., t is a c-approximation of T_0. Now if we return tY, we will ensure that:

$$T_3[(1-\delta)/c] \leq tY \leq T_3[c(1+\delta)] \; .$$

That is, tY is a good estimate of the number of triangles in the graph.

5.3 Analysis

In terms of space, this algorithm needs to maintain $s = \Theta(n)$ samplers. Each sampler needs $O(\log^4 n)$ space, and so we need $O(n \log^4 n$ space.

In terms of the probability of error, we can choose the error parameter ϵ for the $L0$-samplers so that each sampler fails with probability at most $1/(8s) = \Theta(1/n)$. Taking a union bound over the s samplers yield an error probability of at most $1/8$. We also know that the sampling procedures itself fails with probability at most $1/8$. Finally, let us assume that the $L0$-estimator also fails with probability at most $1/8$. This ensures that the total probability of failure is at most $3/8$, i.e., the algorithm succeeds with probability at least $5/8$.

In terms of running time, each edge addition or deletion requires updating $\Theta(n)$ samplers, and each edge update requires updating $n-2$ components of the vector. Hence the update cost for every edge update is $\Theta(n^2 \log^2(n))$. In the end, to estimate the number of triangles, we need to retrieve $\Theta(s)$ samples, and hence the cost is $\emptyset(n \log^2(n))$.

5.4 L0-estimation

The final issue is how we estimate $||v||_0$ using our $L0$-samplers. (There are many alternate and better approaches, but we already have L0-samplers, so let's use them.)

Recall that inside an $L0$-sampler is an array of samplers $S_1, S_2, \ldots, S_{\log n}$ where each sampler S_i receives each index in the vector with probability $1/2^i$. We showed that by choosing $c = 12\ln(4/\epsilon)$, and choosing k to be the largest power of 2 that is less than z/c where z is the number of non-zero indices in the vector, we can ensure with probability at least $\epsilon/2$ that the sampler $S_{\log k}$ is s-sparse, and hence returns a non-zero index.

What about the samplers larger than $S_{\log k}$? Eventually, the sampler receives only zero-indices and hence returns NULL (or error). In particular, let us define ℓ to be the smallest power of 2 larger than z, and consider the sampler $S_{\log \ell+6}$. What is the probability that the vector sent to this sampler has *any* non-zero indices?

For each non-zero index i, the probability that it is included in the sampler is $1/(64\ell)$. Thus, the probability that none of the non-zero indices are included in the sampler is:

$$\begin{aligned} \left(1 - \frac{1}{64\ell}\right)^z &\geq (1 - \frac{1}{64\ell})^\ell \\ &\geq (e^{-2})^{1/64} \\ &\geq 31/32 \end{aligned}$$

(Assume, here, that $\ell \geq 2$.)

Thus, the probability that the sampler $S_{\ell+6}$ returns a non-NULL value is $\leq 1/32$. If you do the calculation, you will find that as you look at the subsequence samplers, the probability decreases geometrically. Thus the probability that any of the samplers $S_{\ell+j}$, for any $j \geq \ell + 6$, return a non-null value is $\leq 1/16$. There is also a probability of $\epsilon/2$ that the samplers fail; by choosing $\epsilon \leq 1/8$, we can ensure that the total probability of failure is at most $1/16+1/16 = 1/8$.

Therefore, we can suggest the following algorithm: let i be the largest index such that sampler S_i returns a non-null value. Return 2^i. We know that the value 2^i will be at least $2^k \geq (z/(2c))$. And we have just shown that it is at most 128. Thus, there is some constant α such that the value returned t is: $z/\alpha \leq t \leq \alpha z$, i.e., it is an α-approximation of $||v||_0$.

Now, in this case, α is quite large. This is not a very good estimator. A more careful analysis (and a more careful choice of the samplers probabilities) would yield a smaller constant. In reality, there are a variety of better techniques for estimating $||v||_0$ that yield better approximation factors much more simply.