7

Graph Summaries

Graphs are a ubiquitous model for complex networks, with applications spanning biology, social network analysis, route planning, and operations research. As the data represented in these applications increases in size, so do the corresponding graphs. This chapter describes summaries that apply to graphs. A graph G is defined by its vertex set V and its edge set E. We assume that V is known, while the edges E can vary. For simplicity, we consider only simple graphs, that is, graphs which are undirected, do not have edge weights, and do not have self-loops.

The graph summaries that we describe tend to have the property that their size is at least proportional to $|V|$, the number of vertices. Although this quantity can still be offputtingly large in many applications, it is the case that there are often hard lower bounds which mean that no summary can exist with size that is asymptotically smaller than $|V|$. Lower bound techniques and some lower bounds for graph problems are discussed in Chapter 10.4.

Summaries differ, depending on what kinds of updates are allowed. Some problems which are straightforward when edges are only added to the graph (such as keeping track of the connected components of the graph) become more complex when edges can be added and removed.

7.1 Graph Sketches

Brief Summary. The Graph Sketch summary allows the connected components of a graph to be found. It makes clever use of the ℓ_0-sampler summary (Section 3.9) applied to inputs derived from graphs. The algorithm stores a number of ℓ_0-sampler summaries for each node. If the graph consists of only edge arrivals, then the problem is much easier: we can keep track of which nodes are in which components, and update this assignment as more information arrives. However, when there can be edge departures as well as arrivals, this simple approach breaks down, and a more involved solution is needed.

The idea behind the Graph Sketch summary is to allow a basic algorithm for connectivity to be simulated. The basic algorithm is to start with each node in a component of its own, then repeatedly find edges from the (current) edge set that connect two different components, and merge these components. If this is performed for all edges in parallel, then a small number of iterations is needed before the components have been found. The insight behind the Graph Sketch is to define an encoding of the graph structure to allow the summary to be built and for this algorithm outline to be applied.

Algorithm 7.1: Graph Sketch: INITIALIZE (r)

```
1 for $j \leftarrow 1$ to $r$ do
2     for $i \leftarrow 1$ to $n$ do
3         $S_{i,j} \leftarrow \ell_0$-sampler.INITIALIZE $(n, 1/n^2)$ ;
```

Operations on the summary. To INITIALIZE a Graph Sketch, we create a set of $r = \log n$ ℓ_0-sampler summaries for each of the n nodes.

Algorithm 7.2: Graph Sketch: UPDATE $((u, v))$

```
1 $a \leftarrow \min(u, v)$;
2 $b \leftarrow \max(u, v)$;
3 for $j \leftarrow 1$ to $r$ do
4     $S_{a,j}$.UPDATE $(a * n + b, +1)$ ;
5     $S_{b,j}$.UPDATE $(a * n + b, -1)$ ;
```

To UPDATE the Graph Sketch with a new edge (u, v), we first encode it as an update to the ℓ_0-sampler summaries for node u and node v. We can assume that u and v are both represented as integers, and we assume that $u < v$ (if not, interchange the roles of u and v). We also assume that we can treat the edge (u, v) as an item that can be processed by the ℓ_0-sampler summaries (for concreteness, we can think of (u, v) as being encoded by the integer $\min(u, v) * n + \max(u, v)$).

We UPDATE the ℓ_0-sampler summaries associated with node u with the item (u, v) and a weight update of $+1$. We also UPDATE the ℓ_0-sampler summaries associated with node v with the item (u, v) and a weight update of -1. Note that it is also straightforward to process edge deletions by swapping the +1 and -1 weights in the UPDATE procedure.

Algorithm 7.3: Graph Sketch: QUERY ()

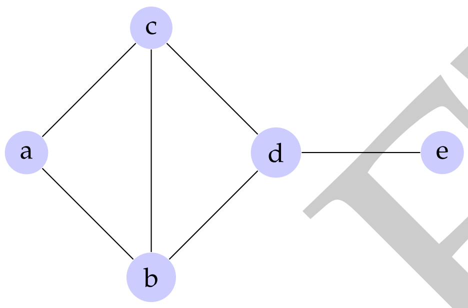
```
1  $\mathcal{C} \leftarrow \emptyset$;
2  for $i \leftarrow 1$ to $n$ do
3      $C_i = \{i\}$ ;
4      $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_i\}$;
5      $T_i = i$ ;
6  for $j \leftarrow 1$ to $r$ do
7      forall $C \in \mathcal{C}$ do
8          $S \leftarrow \ell_0$-sampler.INITIALIZE $(n, 1/n^2)$ ;
9          forall $i \in C$ do
10             MERGE $(S, S_{i,j})$ ;
11         $(u, v) = S$.QUERY () ;
12         forall $w \in C_{T_v}$ do
13             $T_w \leftarrow T_u$ ;
               /* Update the component information   */
14         $C_{T_u} \leftarrow C_{T_u} \cup C_{T_v}$ ;
15         $\mathcal{C} \leftarrow \mathcal{C} \setminus \{C_{T_v}\}$ ;
16 return $\mathcal{C}$;
```

The QUERY procedure is rather involved. We begin by placing each node in a component of its own, with a unique label. We then repeat the following procedure for r rounds. In round j, create a sketch for each component by performing MERGE of the jth sketch of each of the nodes in the component. We then query this sketch to sample an edge. The crux of the process is that, because of the update procedure, the edge sampled is guaranteed to be outgoing from the component (if there are any such edges). This edge is used to combine the component with another. At the end of the rounds, the current set of components is returned as the set of connected components in the graph.

The pseudocode in Algorithm 7.3 introduces some extra notation to describe this procedure. Lines 3 to 5 initialize this by creating n initial components, $C_1 \ldots C_n$, and stores the set of components as $\mathcal{C}$. For convenience of reference, it also instantiates a map from nodes to component identifiers, so that T_u means that node u is part of component C_{T_u}.

The main loop in the pseudocode is over component C in round j. The sketch S is built as the merger of all the sketches of nodes i in component C (line 10), from which an outgoing edge is found (line 11). Based on this edge (u,v), we assign all the nodes in the component of v to be in the component of u (line 13). We then combine the components of u and v (line 14), and remove the component of v from the set of components C.

Example. For an example, we show a small graph, and study the way that its edges are encoded before being placed into the sketch. Consider the graph below:



The initial encoding of the edges can be described in the following table:

	(a,b)	(a,c)	(a,d)	(a,e)	(b,c)	(b,d)	(b,e)	(c,d)	(c,e)	(d,e)
a	+1	+1								
b	-1				+1	+1				
c		-1			-1			+1		
d						-1		-1		+1
e										-1

Each node is associated with one row of the table that encodes its neighbors. Here, we index the rows by the identity of the edges (in the pseudocode description, we convert these to use a canonical integer representation). Node b is linked to three neighbors, a, c, and d. Since a comes before b in the (alphabetic) ordering of the node, edge (a,b) is represented by a -1 in the table, while edges (b,c) and (b,d) are encoded as +1 in b's row in the table. Observe that each edge is thus represented twice, with a +1 for one occurrence and a -1 for the other. Note that the table itself is not stored explicitly; rather, the summary keeps an instance of an ℓ_0-sampler for each row.

Now suppose that after one iteration of the algorithm, we have chosen to merge nodes a and c into one cluster, and b, d, and e into another. We now obtain the resulting table for the clusters:

	(a,b)	(a,c)	(a,d)	(a,e)	(b,c)	(b,d)	(b,e)	(c,d)	(c,e)	(d,e)
{a, c}	+1	0			-1			+1		
{b,d,e}	-1				1	0		-1		0

The row {a, c} in the new table is the result of summing the rows *a* and *c* from the first table (in the summary, we have obtain an ℓ_0-sampler of this sum). Here, we write 0 to denote where a +1 and −1 have co-occurred and annihilated each other. Similarly, the row {b, d, e} is the sum of the three rows *b, d, e* from the first table. Observe that the only edges which remain in this representation are those that cross between the two components: (a, b), (b, c) and (c, d).

Further Discussion. The correctness of the algorithm depends critically on the encoding of edges, and the properties of the ℓ_0-sampler structure. Consider a collection of nodes C and their corresponding merged ℓ_0-sampler structure. Observe that if an edge (u, v) connects two nodes $u \in C, v \in C$ then the contribution of this edge in the sketch is exactly 0: it is represented with +1 in the sketch of u and −1 in the sketch of v, so when these are merged together, the net contribution is zero. Therefore, this 'internal' edge cannot be sampled from the merged sketch. Hence, only 'outgoing' edges (where one node is in C and the other is not in C) can be drawn from the sketch.

The rest of the analysis then reduces to analyzing the algorithm that picks one outgoing edge from each component in each round, and merges the pairs of components. The number of components must at least halve in every round (excluding any connected components which have already been fully discovered), hence the number of rounds is at most $\log n$ since we start with n components (each containing a single node).

It remains to argue that, over all the accesses to sketches, the chance of failing to draw a sample from any is very low. From the above analysis, it follows that over the rounds we make no more than $n + n/2 + n/4 + \ldots < 2n$ calls to the QUERY routine for an ℓ_0-sampler structure. If we set the failure probability of each of these to be at most $1/n^2$, then the chance that there is any failure across all the whole operation of the Graph Sketch structure is still very small, $2/n$, by appealing to the union bound (Fact 1.6).

It is important for the correctness of the algorithm that independent sketches (using different random hash functions) are used in each round. This ensures that we can correctly analyze the probability of success. For intuition, observe that if it were possible to use a single set of ℓ_0-sampler structures (one for each node), then we could use them to extract incident edges on each node, delete these edges from the summaries, and repeat until we have found all edges in the graph. That would allow the recovery of $O(n^2)$ edges from n summaries of small size – intuitively this should be impossible! Indeed, this intuition can be formalized, and so we cannot hope to "recycle" the summaries so aggressively.

History and Background. The idea of graph sketches was introduced by Ahn, Guha and McGregor in [7]. They used variations of this idea to also establish k-connectivity of graphs (where the graph remains connected even up to the removal of k nodes) and bipartiteness (by expressing bipartiteness in terms of the number of connected components in a derived graph). However, since we need to keep multiple ℓ_0-sampler summaries for all n nodes, each of which is typically kilobytes in size [61], the space cost for this summary is quite large.

Similar summary ideas have been used for the problem of dynamic graph connectivity in poly-logarithmic time [147]. Here, it is required to maintain an explicit representation of the connected components capable of answering whether a pair of nodes u and v are in the same component quickly.

7.2 Spanners

Brief Summary. A k-spanner of a graph G is a subgraph H (both defined over vertices V) so that for any pair of nodes u and v

$$d_G(u, v) \leq d_H(u, v) \leq k d_G(u, v).$$

That is, every distance in G is stretched by a factor of at most k in its Spanner. Spanners can be relatively easy to build, but come with some limitations. The first is that the values of k tend to be moderate constants, say 3 or 5 – whereas, in many cases we would prefer that k be close to 1 to preserve distances as much as possible. However, it is only with these larger k values that we can guarantee that the Spanner will be smaller than the original graph G. The second is that the Spanner we describe does not have a MERGE operation – they can only be built incrementally by a sequence of UPDATE operations. This restricts their applicability.

Algorithm 7.4: Spanner: INITIALIZE (V, k)

```
1 $H \leftarrow (V, \emptyset)$;
2 Record the value of $k$ ;
```

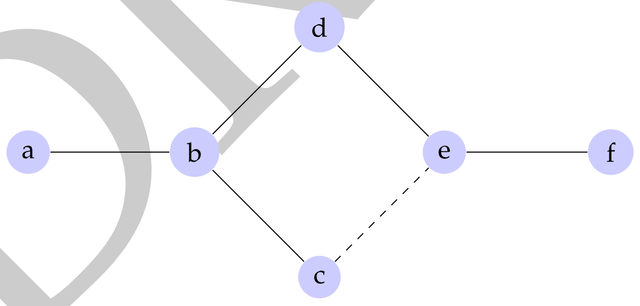
Algorithm 7.5: Spanner: UPDATE $((u, v))$

```
1 if $d_H(u, v) > k$ then
2     $E \leftarrow E \cup \{(u, v)\}$ ;
```

Operations on the summary. To INITIALIZE a new Spanner based on stretch parameter k, we create an empty graph on the vertex set V, and store k. To UPDATE a Spanner with a new edge (u, v), we consider whether it is necessary to add the edge. If the distance between u and v in the current version of the stored graph H is at most the stretch parameter k, then we do not need to store it. Otherwise, we do need to retain this edge to keep the promise, and so it is added to H. This is shown in Algorithm 7.5.

The queries that the spanner supports are distance and reachability queries. Given a pair of nodes (u, v), we approximate their distance in the input graph G by returning their distance in the Spanner graph H, as $d_H(u, v)$.

Example. Consider the following graph, processed by the algorithm with parameter $k = 3$.



Suppose we have processed all edges up to the dashed edge (c, e). All edges up to this point are retained in the summary, as none can be dropped without disconnecting the graph. For edge (c, e), the graph distance between nodes c and e is 3 — via c, b, d, e. So the edge (c, e) is not retained in the summary.

Now any distance in the graph is preserved up to a factor of (at most) 3. The distance between c and f is 2 in the original graph, but is 4 in the spanner, since the direct edge (c, e) is replaced by the path c, b, d, e.

Further Discussion. There are two steps to arguing that H provides a spanner of bounded size. First, we argue that the distances in H only stretch by a factor of k. Consider some path in G, which is a sequence of edges. In the worst case, none of these edges are retained in H. But if this is the case, then we know that for each edge that was discarded, there is a path in H of k edges. So the path in G is replaced by one at most k times longer.

The second part is to argue that the size of H does not grow too large. For this, we rely on some facts from the area of "extremal graph theory", regarding the graphs which lack some particular subgraph [31]. By construction, graph H has no cycles of size $k + 1$ (or smaller) — since the last edge to arrive in the cycle would be dropped by the summary construction algorithm. Such graphs can have at most $O(n^{1+\frac{2}{k+1}})$ edges. For moderate values of k, say $k = 3$, we get guarantees that the size of H is certain to be much smaller than the theoretical maximum of n^2 possible edges in G.

Implementation Issues. A key implementation requirement is to be able to find $d_H(u, v)$ online as new edges are added to the Spanner. Efficient algorithms and data structures for this problem are beyond the scope of this volume, but this consideration has played into other algorithms for Spanner construction.

History and Background. The simple algorithm for Spanner construction described above is due to Feigenbaum *et al.* [98]. There has been much subsequent work on the problem of maintaining Spanners as edges are added. In particular, Elkin shows how to improve the time to process each edge with a randomized algorithm [93].

7.3 Properties of Degree Distributions via Frequency Moments

The degree distribution of a graph describes the number of nodes that have a particular degree. For many large graphs of interest (representing social networks, patterns of communication etc.), the degree distribution can be approximately described by a small number of parameters. For example, in many cases, the distribution is heavily skewed: a small number of nodes have high degree, while the majority have very low degree (the long tail). Such behavior is often modeled by a power-law distribution: the fraction of nodes with degree d is taken to be proportional to d^{-z}, for a parameter z typically in the range 0.5 to 2.

Since the degree distribution can be captured as a vector, indexed by node identifiers, it is natural to apply many of the summaries seen previously to describe its properties. For example, we might naturally find the identifiers of the nodes with highest degree using summaries like SpaceSaving or Count-Min Sketch (Chapter 3). Improved results are possible by mixing different sampling approaches: a KMV sample of the nodes to describe the behavior of the "head" of the frequency distribution, and a uniform RandomSample of the nodes to describe the behavior of the "tail" [205]. Tracking the sum of squares of degrees, i.e., F_2 via AMS Sketch, may allow the estimation of parameters of models such as a power-law distribution.

These analyses assume that the graph is presented so that each edge is observed only once. In some settings, we may see each edge multiple times (e.g., seeing multiple emails between a pair of communicating parties), but only wish to count it once. We can track the total number of distinct edges seen via HLL. For more complex queries, we can combine various of our summaries. For example, we can estimate the underlying degree of each node u, given by the number of distinct neighbors v of u seen among the edges. This can be done by nesting a distinct counting summary (e.g., HLL) within a frequent items structure (e.g., Count-Min Sketch). This approach is discussed at more length in Section 9.4.3.

7.4 Triangle Counting via Frequency Moments

Many analyses of graph-structured data rely on the notion of the triangle: a complete subgraph on three nodes. Detecting and counting such local structure can be hard when the graph to be analyzed is large, and possibly broken into multiple distributed pieces. Nevertheless, there has been a large amount of effort directed at problems to do with triangles: sampling a representative triangle, approximately counting the number of triangles, and so on.

Here we describe one method for estimating the number of triangles in a graph based on summaries described earlier. Given a graph $G = (V, E)$, each triangle is defined by a triple $(u, v, w) \in V^3$ such that $(u, v) \in E$, $(v, w) \in E$, and $(u, w) \in E$. We proceed by converting each edge into a list of the possible triangles that it can be a member of: from edge (u, v), we generate (u, v, w) for all $w \in V$. We then consider a vector x that encodes the total number of occurrences of each possible triangle: $x_{(u,v,w)}$ counts the number of times that possible triangle (u, v, w) is listed.

If each edge is seen exactly once in the description of G, then the number of triangles in G is the number of entries that are 3 in the vector x. A value of 3 can arise at index (u, v, w) only if all three edges (u, v), (v, w) and (u, w) are seen in the input. Tracking x exactly will require a lot of storage ($O(n^3)$ space to describe all possible triangles), and so we will use summaries. We have summaries that can approximate $F_2(x)$, i.e., $\sum_i x_i^2$, such as the AMS Sketch. We can also approximate $F_0(x)$, i.e., $\sum_{i:x_i \neq 0} 1$, such as the HLL. Lastly, we can compute $F_1(x)$ directly, as the sum of all entries in x.

Since the possible values in x are restricted to $\{0, 1, 2, 3\}$, we can use an algebraic trick to count only the entries that are 3. Observe that the polynomial $p(y) = 1 - 1.5y + 0.5y^2$ behaves as follows

$$p(1) = 0 \qquad p(2) = 0 \qquad p(3) = 1$$

Now observe that $P(x) = F_0(x) - 1.5F_1(x) + 0.5F_2(x)$ is equivalent to counting 0 for each zero entry of x, and applying polynomial p to each non-zero entry of x. Consequently $P(x)$ counts the number of triangles in G exactly.

By using summaries, we are able to approximate $P(x)$. We obtain an error of $\epsilon(F_0(x) + F_2(x))$ in our estimate — note that we can maintain $F_1(x)$ exactly with a counter. There's no immediate guarantee that the error term $\epsilon(F_0(x) + F_2(x))$ is related to the number of triangles T, and indeed, one can create graphs which have no triangles, but for which $F_0(x)$ and $F_2(x)$ are quite large. However, we can easily show that $F_0(x) \leq F_1(x)$ and $F_2(x) \leq 3F_1(x)$, and that $F_1(x) < mn$, where $m = |E|$ and $n = |V|$. Consequently, we obtain an error guarantee of at most $4\epsilon mn$.

A limitation of this approach is that the step of generating all possible triangles for edge (u, v) to feed into summaries would require $O(n)$ UPDATE operations if done explicitly. To make this efficient, we must adopt summaries for F_0 and F_2 that are efficient to update when presented with an implicit list of updates. That is, edge (u, v) implies a list of edges (u, v, w) for all $w \in V$. Such summaries are called *list-efficient*.

History and Background. This approach to triangle counting is due to Bar-Yossef, Kumar and Sivakumar [19]. This work introduced list-efficient summaries for F_0. They adapt prior work of Gilbert *et al.* on estimating F_2 to be list-efficient for the lists of triangles that are generated [112]. Subsequent work has aimed to provide more efficient list-efficient summaries [194, 216]. Other approaches to counting triangles are based on sampling and counting, and so tend to imply summaries that can perform UPDATE but not MERGE operations (i.e., they process streams of edges). Some examples include the work of Jowhari and Ghodsi [143], Buriol *et al.* [42], Pavan *et al.* [193], and most recently McGregor *et al.* [173].

7.5 All-distances Graph Sketch

The all-distances graph sketch (ADS) keeps information about the neighborhood of every node v in a graph G. It allows us to approximate functions based on the number of nodes at different distances from v. For example, a basic question would be "how many nodes are within distance d from node v?". More generally, it can also answer questions such as "how many red nodes are within distance d from node v" (if each node has a different color), or "compute the sum of the reciprocals of the distances of all nodes from v". Formally, we can compute arbitrary functions of the form $\sum_u f(u, d(u, v))$. The sum is over all nodes u in the same connected component as v, and allows the function f to specify a value based on the information of node u (e.g., color) and the graph distance between u and v, given as $d(u, v)$.

The summary can be instantiated based on a number of different summaries for counting distinct items, such as KMV and HLL. For concreteness, we describe a version that builds on KMV (Section 2.5). Recall that the KMV structure keeps a summary of a set by applying a hash function h to each member of the set, and retaining only the k elements which achieve the k lowest hash values. The ADS extends KMV by keeping additional information on graph distances as well. Assume for now that we have convenient access to $d(u, v)$ for all nodes u and v in the graph. For simplicity, let us assume that all distances are distinct (this can be achieved by breaking ties based on node ids, for example). Now we keep node u in the ADS structure for v if its hash value (under hash function h) is among the k smallest for all nodes w whose distance from v is at most $d(u, v)$.

Algorithm 7.6: ADS:INITIALIZE (v,k)

```
1 Pick hash function $h$, and store $k$;
2 Initialize list $L_v = \emptyset$;
```

The INITIALIZE operation for a node v (Algorithm 7.6) is almost identical to that for KMV: it picks a hash function h mapping onto a range $1 \ldots R$, and creates an empty list that will hold information on nodes.

Algorithm 7.7: ADS: UPDATE $(u, d(u, v))$

```
1 if $u \notin L_u$ then
2     $L_v \leftarrow L_v \cup \{(u, d(u, v), h(u))\}$;
3     if $|\{w \in L_v | d(w, v) \leq d(u, v)\}| > k$ then
4         $x \leftarrow \arg\max_{w \in L_v, d(w,v) \leq d(u,v)} h(w)$ ;
5         Remove $x$ from $L_v$ ;
```

The UPDATE operation to add information about a node u to the summary for node v (Algorithm 7.7) automatically inserts the new node into the data structure for v. It then checks if the condition on the number of nodes at distance at most $d(u, v)$ is violated, and if so, deletes the node within that distance with the maximal hash value.

The consequence of this definition is to draw a sample (via the randomly chosen hash function h) where the inclusion probability for a node depends on its distance from v. The k closest nodes to v are certain to be included. Nodes very far from v have a lower chance to be included, essentially a k/n chance, when there are n nodes in the (connected) graph G. For the node which has the ith furthest distance from v, the chance that it is kept in the ADS is

$\min(1, k/i)$. Then, we can quickly see that the expected size of the ADS is $k + \sum_{i=k+1}^{n} k/i \le k \ln n$. That is, a moderate factor than the $O(k)$ size of a regular KMV summary.

To appreciate the power of the ADS, we first consider the query to estimate the number of nodes whose distance is at most d from v. In the ADS data structure, we extract all stored nodes that meet this distance restriction. Note that, provided that in the full graph G there are at least k such nodes, then we will recover at least k nodes from the summary. This follows by definition of which nodes are held in the summary. Furthermore, we can consider what would have happened if we had built a KMV summary applied to only those nodes which meet the distance restriction, using the same hash function h. Then we would have retained at least k of the nodes extracted from the ADS. That is, we have all the information necessary in the ADS in order to extract a KMV summary of this subset of the input. Consequently, we can apply the QUERY procedure for KMV, which is based on the hash values of the retained elements to estimate the cardinality of the set from which they were drawn. This provides a $(1 \pm \epsilon)$ guarantee for the number of nodes within distance d, provided that $k = O(1/\epsilon^2)$. In the case when there are fewer than k nodes at distance d from v, then the ADS is required to keep all of them, and so we obtain the exact result.

Algorithm 7.8: ADS: QUERY (v, P,d)

```
1 $Q \leftarrow \{u | u \in L_v, d(u,v) \le d\}$;
2 if $|Q| \le k$ then
3     $r = k$
4 else
5     $v_k \leftarrow \max_{w \in Q} h(w)$ ;
6     $r \leftarrow (k-1) * R/v_k$;
7 return $r * |\{u | u \in L_v, d(u,v) \le d, P(u) = \mathbf{true}\}| / |Q|$
```

This approach forms the basis of answering more general queries. For queries such as "how many red nodes are within distance d of v", we again extract all nodes in the ADS summary that are within distance d. We use this to estimate the total number of nodes at distance d. We then inspect the nodes extracted, and compute the fraction that meet the predicate (colored red), and give our estimate of the number as this fraction times the total number within distance d. The error in this estimate is proportional to $\epsilon|N_d|$, where N_d is the number of nodes within distance d. This follows by analogy with applying predicates to the KMV summary, discussed in Section 2.5. This show in Algorithm 7.8, to apply predicate P and distance bound d to the summary of node v, and estimate the number of nodes meeting this predicate within distance d from v.

For more general functions $F(v) = \sum_u f(u, d(u, v))$, we can proceed by (notionally) iterating over every node u in the graph. For any given node, we can use the ADS to estimate its contribution to the approximation of $F(v)$. This is given by $f(u, d(u, v))$, scaled by the estimate for the number of nodes within this distance. Observe that for nodes w not stored in the ADS summary, we have a contribution of 0 to the estimate. Therefore, it suffices to just iterate over every node that is stored in ADS, and compute its contribution to the estimate.

It remains to discuss how to build an ADS summary for nodes in a graph without exhaustively computing the all-pairs shortest path distances. Assume that we want to compute the ADS summary for all nodes in G. We start by instantiating an empty summary for each node. We fill each summary by creating a "self-referential" entry for node v at distance 0 from itself. Then each node v can follow a 'gossip' style algorithm: every time it hears about a new node distance pair (u, d), it can test whether it should be included in its current ADS (possibly evicting some pair that now no longer meets the criterion for inclusion). If (u, d) is added to v's ADS, then v can inform all its neighbors w about the new node, at distance $d + d(v, w)$. Note that a node v might hear about the same node u through different paths. If so, v can ensure that it only retains information about u with its shortest distance. It is possible to see that this process will converge to the correct result, after a number of "rounds" proportional to the diameter of the graph. It's less immediate how to bound the cost of this procedure, but we can observe that we do not expect many nodes to propagate very far through the graph, due to the definition of the ADS summary. The total number of operations to insert nodes into ADS structures across the whole graph can ultimately be bounded by the product of the size of each structure times the number of edges, m, as $O(km \log n)$.

Example. We give a small example of the QUERY procedure for ADS with $k = 3$. Consider the following set of nodes, where for each node we list its distance from the node v, and its hash value (an integer in the range 1 to 20).

Node id	Distance from v	Hash value
a	1	19
b	2	5
c	3	12
d	4	8
e	5	15
f	6	3
g	7	9
h	8	2

Then for $k = 3$ we would retain the set of nodes $\{a, b, c, d, f, h\}$. We retain d since it has a lower hash value than a which has closer distance. We omit e since there are already 3 nodes with lower hash values at closer distance. Likewise, g is dropped since b, d and f have lower hash values and smaller distances.

To estimate the number of nodes with distance 4 or less, we can extract the k nodes in the ADS summary that form a KMV summary: that is, b, c, and d (a would be dropped from the KMV summary). Our estimate is given by $(k - 1) * R/v_k$ where $R = 20$ is the range of the hash function and $v_k = 12$ is the kth highest hash value (see Algorithm 7.8 and discussion for the KMV QUERY operation in Section 2.5). This gives the estimate of 3.33 elements, which is tolerably close to the true result of 4. Note that we could have made more use of the information available, since we also have knowledge of a and its hash value. This is discussed by Cohen [53], who defines an improved estimator based on the "inverse probability" of each node to be included in the summary.

History and Background. The central ideas for the ADS are due to Cohen [52], who introduced the summary to estimate the number of nodes reachable from a node, and the size of the transitive closure in directed graphs. Our presentation follows a later generalization by Cohen [53] that goes on to consider different constructions of the ADS and tight bounds on their estimation accuracy based in the notion of “Historic Inverse Probability (HIP)” estimators.