Algorithms

Lecture Notes — Randomized Algorithms

Imdad ullah Khan

Contents

1 Introduction

We have seen various algorithm design paradigms so far: greedy, divide-and-conquer, dynamic programming and network flows. We have also used some of these types of algorithms to cope with NP-HARD problems, compromising on different aspects such as polynomial runtime (Intelligent Exhaustive Search), exact solution (Approximation & Heuristic Algorithms), and solving for all cases (Special Cases) and parameters (Fixed Parameter Tractability).

One thing that all these algorithms have in common is that they are *deterministic*, i.e. they always produce the same output for the same input. However, surprisingly, some of the smartest and fastest algorithms that exist depend on odds or *chance*, i.e. some specified decisions in the algorithm are based on the outcome of the toss of a random coin. Such algorithms are called *randomzied* algorithms. So how is randomness incorporated in the operation of these algorithms?

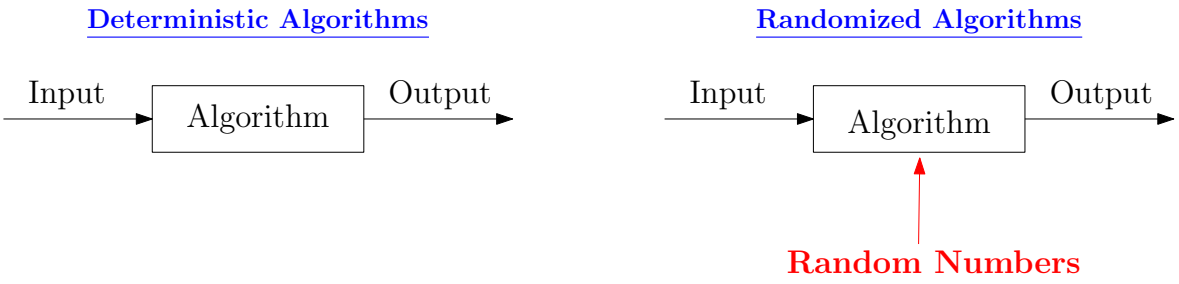


Figure 1: Deterministic vs. Randomized Algorithms

As shown in Figure 2, a randomized algorithm receives, in addition to the input, a random number stream to make decisions during its execution. As a result, randomized algorithms may output different results on the same input across different runs. When designing a randomized algorithm, the aim is to have good average-case (or expected) behaviour, which means that we should get exact answers, or answers close to the correct one, in a small runtime with high probability. Often, randomized algorithms are simple and elegant, but there is a high probability of producing an output which is correct to some acceptable degree. Another advantage of randomized algorithms is that they require less execution time or space than the best deterministic algorithm.

1.1 Monte Carlo and Las Vegas Algorithms

We can broadly classify randomized algorithms into two types: Las Vegas and Monte Carlo algorithms. Monte Carlo algorithms introduce randomness in the solution, i.e. they are guaranteed to run in a fixed time but are expected to output a correct an-

swer with some, usually high, probability. On the other hand, Las Vegas algorithms introduce randomness in the runtime, i.e. they are guaranteed to output the correct answer but the runtime is expected to be small with high probability. An interesting thing to note is that we can always convert a Las Vegas algorithm into a Monte Carlo algorithm by stopping the algorithm after a certain point (fixed time). However, there is no known method for the reverse conversion and it needs efficient verification.

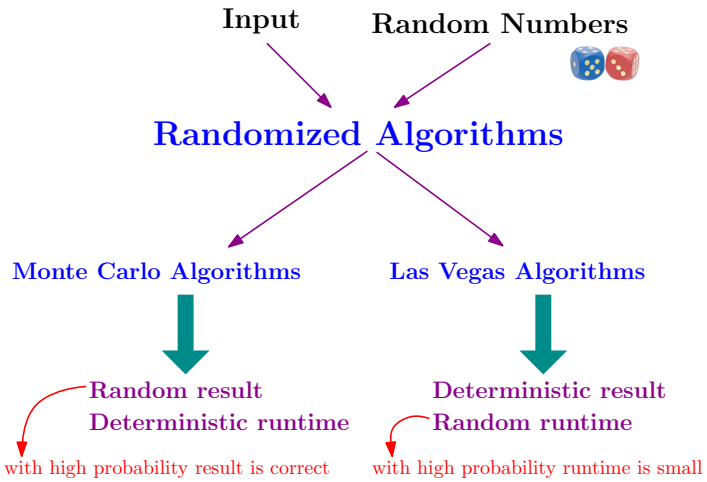


Figure 2: Monte Carlo and Las Vegas Randomized Algorithms

We illustrate the differences in Deterministic and Monte Carlo and Las Vegas Randomized algorithms through a simple problem.

Input: An array A with $n/4$ 1's and $3n/4$ 0's

Output: An index k such that $A[k] = 1$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	1	0	0	0	0	0	0	1	0	0	1	1	0

We give a deterministic algorithm and a Monte Carlo and Las Vegas randomized algorithm for the problem and illustrate their qualitative and runtime analysis.

Algorithm Deterministic

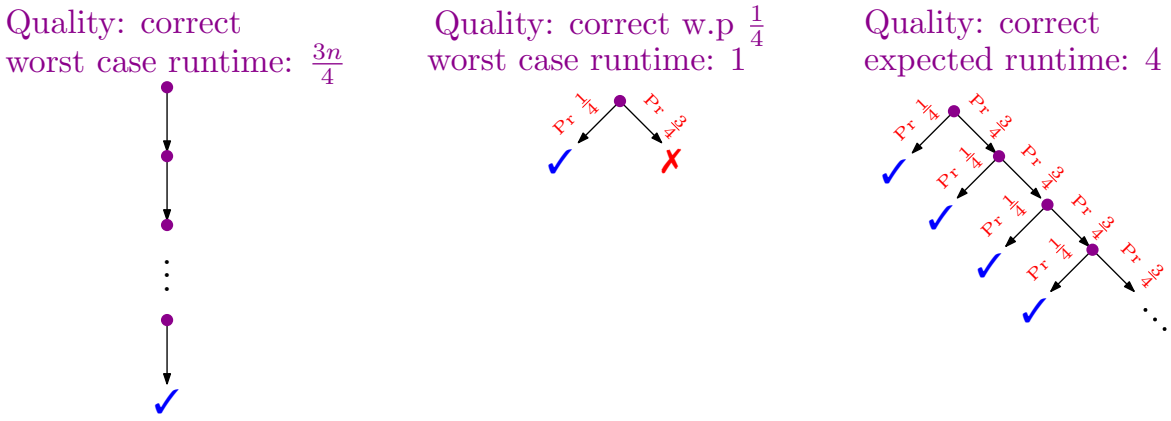
```
k ← 0
for i = 1 → n do
    if A[i] = 1 then
        k ← i
        return k
```

Algorithm Monte Carlo

```
k ← RANDOM(1 ··· n)
return k
```

Algorithm Las Vegas

```
k ← 1
while A[k] ≠ 1 do
    k ← RANDOM(1 ··· n)
return k
```



We will study, among others, a Monte Carlo algorithm for the Min-Cut problem, and Las Vegas algorithms for sorting and closest-pair problems.

2 Probabilistic Analysis: Quick-Sort

Recall the recursive QUICK-SORT algorithm outlined in Algorithm 4.

Algorithm 4 Sorting A using PARTITION

```
1:  function QUICKSORT(A)
2:      if |A| ≤ 1 then
3:          return A
4:      z ← A[1]                                              ▷ pivot z
5:      PARTITION(A, z)
6:      r ← RANK(z, A)
7:      QUICKSORT(A[1 ... r − 1])
8:      QUICKSORT(A[r + 1 ... |A|])
9:  function PARTITION(A, z)
10:     i ← 1     j ← |A|
11:     r ← RANK(A, z)
12:     while i < j do
13:         while doA[i] < z
14:             i ← i + 1
15:         while doA[j] > z
16:             j ← j − 1
17:         if i ≠ r AND j ≠ r then
18:             SWAP(A[i], A[j])
```

As shown in Figure 3, the PARTITION procedure rearranges the input array such that all elements less than z are to its left, and all elements greater than z are to its right. The process is recursively repeated for both sub-arrays, to the left and right of z, as shown in Figure 4.

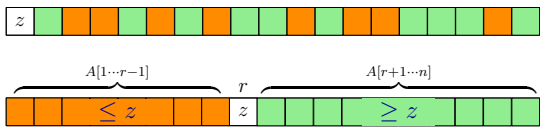


Figure 3: Partitioning A around pivot z

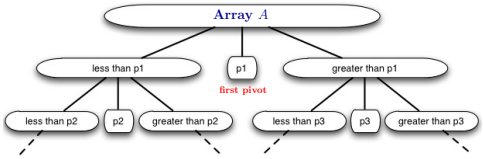


Figure 4: Recursion Tree of QUICK-SORT

Let $T(n)$ be the runtime of QUICKSORT on $|A| = n$. The worst case for the runtime is when the pivot is always the minimum or maximum of A, leading to a completely skewed recursion tree, as shown in Figure 5, from which we do not gain any benefit of the divide-and-conquer approach.

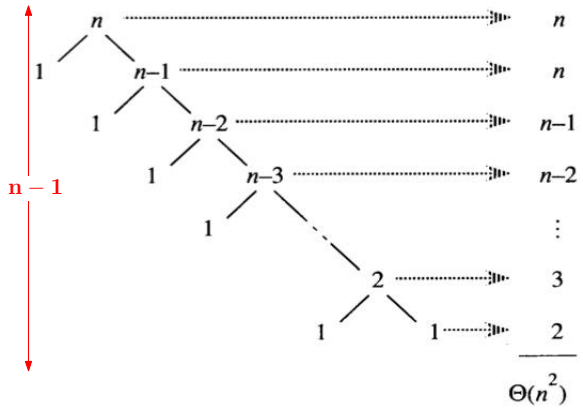


Figure 5: Worst case of QUICK-SORT when $z = \text{MIN}(A)$

The recurrence relation for $T(n)$ in the above case is:

$$T(n) = \begin{cases} T(n-1) + T(0) + O(n) & \text{if } n > 1 \\ 1 & \text{if } n \leq 1 \end{cases}$$

Therefore, in the worst case, $T(n) = O(n^2)$.

On the contrary, the best case of QUICKSORT is when the pivot is always the medium of the array so the left and right sub-arrays after partition are balanced. In this case, the recurrence relation for $T(n)$ is:

$$T(n) = \begin{cases} 2T\left(n/2\right) + O(n) & \text{if } n > 1 \\ 1 & \text{if } n \leq 1 \end{cases}$$

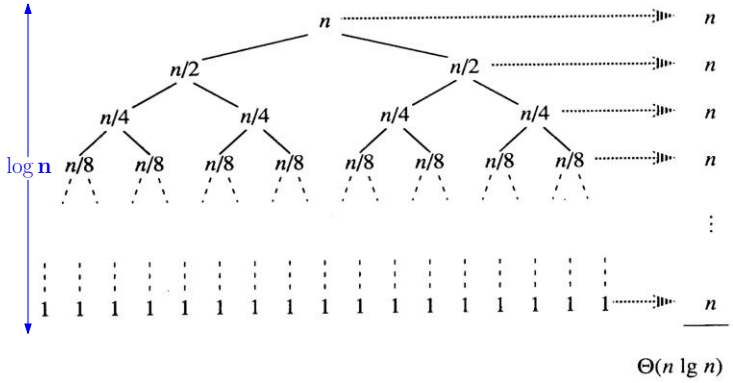


Figure 6: Best case of QUICK-SORT when $z = \text{MEDIAN}(A)$

Therefore, in the best case, $T(n) = O(n \log n)$.

We know the extremes, best and worst case, of runtime performance of QUICKSORT, but for practical purposes, the more feasible question to ask is that what is the *average case* running time of QUICKSORT? But what do we average over? We have or assume knowledge about the distribution of the input, and so we average over the distribution considering how probable each element in the input is to appear. When probability is used in the analysis of a deterministic algorithm, it is called probabilistic analysis.

For the probabilistic analysis of QUICKSORT, we assume all permutations of n numbers in A are equally likely, i.e. ranks of numbers in A is a uniform random permutation of $[1 \cdots n]$. We make a few observations about the QUICKSORT algorithm first.

Note that an element of A can be chosen as pivot at most once, since all subsequent processing is done on the two sub-arrays. Furthermore, elements of A are compared to pivots only as elements are only compared with z in the PARTITION function, and no comparison is made in the outer recursive function. From the previous two statements, we can conclude that a pair of elements of A are compared only when one of them is a pivot, and consequently, a pair is compared at most once. Since comparisons always involve the pivot, after a comparison is made, the two elements always go to different parts. We use these insights to obtain a bound on the expected number of comparisons made.

Let the sorted order of elements of A be $z_1, z_2, \ldots, z_n$. We define an indicator random variable X_{ij} as:

$$X_{ij} = \begin{cases} 1 & \text{if } z_i \text{ is compared with } z_j \\ 0 & \text{else} \end{cases}$$

Note that the comparison between z_i and z_j can be at any time of the execution and not in a specific call. Thus, the total number of comparisons X made during the execution of the algorithm can be obtained by summing over all pairs.

$$X = \sum_{i=1}^{n} \sum_{j=1}^{n} X_{ij}$$

Thus, by linearity of expectation:

$$E(X) = E\left[\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E\left[X_{ij}\right]$$

To find the expected value of X_{ij}, we define Z_{ij} to be the set of elements between z_i and z_j (inclusive). Thus, $|Z_{ij}| = j - i + 1$. Now consider the sequence Z_{ij} : $z_i, z_{i+1}, \ldots, \ldots, z_j$. Initially, all the elements in the sequence Z_{ij} are in the same array A. They split only when some z_k for $i \leq k \leq j$ is selected as the pivot. Recall that z_i and z_j are compared only if they are in the same (sub)array and either z_i and z_j is selected as the pivot. If the first pivot in Z_{ij} is other than z_i and z_j, then Z_{ij} is split, z_i and z_j never get compared, and $X_{ij} = 0$. Therefore, $E[X_{ij}] = Pr[z_i \text{ or } z_j \text{ is the first among } Z_{ij} \text{ chosen as pivot}]$. What is the probability of z_i or z_j being chosen as a pivot first among Z_{ij} before any z_k?

The probability that z_i is chosen as the pivot first among Z_{ij} is $1/j-i+1$. Thus, the probability of comparison is $2/j-i+1$ i.e. sum the probability of z_i being chosen, and of z_j being chosen as the pivot first. Therefore,

$$E(X_{ij}) = \frac{2}{j - i + 1}$$

Substituting $k = j - i$ in the equation for $E(X)$, we get that:

$$E(X) = \sum_{i=1}^{n} \sum_{k=1}^{n-i} = \frac{2}{k+1} < \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{2}{k} \leq 2n \log n.$$

Therefore, the average-case runtime of QUICKSORT is $O(n \log(n))$.

Since the input array can not be guaranteed to be randomly ordered, we permute the array A to make it a random permutation. In fact, generating a random permutation is an interesting exercise. The simple way using an auxiliary array to randomly select an element from A and copy it to the next empty position in an auxiliary array and remove the element from A. However, the time complexity of this algorithm is $O(n^2)$. A smarter approach, which avoids the use of extra space, and can permute the array in $O(n)$ is the Fisher-Yates shuffle algorithm, which assumes that we can generate a random number between 0 and n in $O(1)$. The key idea is to begin at the last element in A and swap it with the element at a randomly selected index in A, including the last element. The process is repeated for the array from the first to the second-last element, and so on, decrementing the array size by 1, until the first element is reached.

We can also reduce the likelihood of the worst case occurring by taking the median of 3 or 4 elements as the pivot. If the pivot is the median of the whole array, then the best, worst and average case time complexity is $O(n \log(n))$.

2.1 Randomized Quick Sort

Instead of randomly permuting the array A for QuickSort as above, we can also choose the pivot randomly, instead of always taking the first element as the pivot as shown in Algorithm 5.

Algorithm 5 Randomized QuickSort

```
function RAND-QUICKSORT(A)
    if |A| ≤ 1 then
        return A
    randIndex ← RANDOM(1, |A|)
    z ← A[randIndex]
     PARTITION(A, z)
    r ← RANK(z, A)
    RAND-QUICKSORT(A[1 ... r − 1])
    RAND-QUICKSORT(A[r + 1 ... |A|])
```

By partitioning around a random element, the runtime is independent of the input order. Therefore, no assumptions need to be made about the distribution of the input and no specific input results in the worst-case behavior. Rather, the worst case (runtime) depends on the sequence of random numbers.

The probabilistic analysis we carried out above was to find the average runtime over all inputs (of length n) of the deterministic QUICKSORT algorithm. For the RANDOMIZED-QUICKSORT algorithm, we find the expected runtime, i.e. the expected value of the runtime random variable of a randomized algorithm. Expected runtime can be understood to effectively be an 'average' of run-time over all sequences of random numbers. The analysis of Randomized-QuickSort is exactly the same as the probabilistic analysis we discussed above, using indicator random variables.

3 Randomized Selection

For the QUICKSORT algorithm, we saw that we can achieve the best case runtime if we repeatedly select the median of each (sub)problem as the pivot to partition around. This is only one case where efficiently finding the median of an array is important. Likewise, for numerous applications, we may need to select the minimum, maximum or the k^{th} smallest element (called the k^{th} order statistic) in an array.

PROBLEM 1 (Select(S, k) problem)**.** *: Find the k^{th} smallest element in S.*

In the case of a sorted array with distinct elements, SELECT can be done in $O(1)$. This also gives us a simple algorithm for any array: first sort the array of distinct elements and then select the element at the k^{th} index. However, such deterministic sorting algorithms take $O(n \log n)$ time.

We discuss a randomized algorithm for the SELECT problem, i.e. select the k^{th} smallest element in an array S. Assume S has only distinct elements. For a randomly chosen $z \in S$, partition S into S_L ($< z$), S_z ($= z$) and S_G ($> z$) as shown in Figure 7.

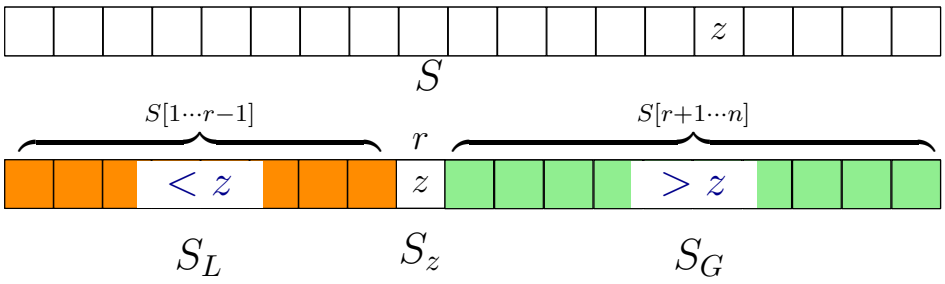


Figure 7: In each sub-problem, choose a random z to partition around until z is the k^{th} order statistic

Subject to the choosing z, the randomized algorithm is given by the following recurrence:

$$\text{SELECT}(S,k) = \begin{cases} \text{SELECT}(S_L, k) & \text{if } k \leq |S_L| \\ z & \text{if } k = r = |S_L| + 1 \\ \text{SELECT}(S_G, k - |S_L| - 1) & \text{if } k > |S_L| + 1 \end{cases}$$

We analyze the runtime $T(n)$ of the above randomized algorithm for SELECT given an input S. Let $|S| = n$. Then,

$$T(n) = T(\text{MAX}\{r, n - r - 1\}) + \Theta(n)$$

We want z to be the median, or close to the median, of S to find the required value efficiently. However, since we choose z randomly, it may be far from the median, resulting in an imbalanced partition. In the wort case, $T(n) = \Theta(n)$. However, the expected runtime $E[T(n)] \leq cn$. This can be easily proven by induction.

Recall that the rank of an element x in an array is the number of elements smaller than than x. Since S contains distinct elements, $Pr[rank(z) = r] = 1/n$. Then,

$$E[T(n)] = n + \sum_{r=1}^{k} T(r) Pr[rank(z) = r] + \sum_{r=k+1}^{n} T(n-r) Pr[rank(z) = r]$$

$$= n + \frac{1}{n}\left[\sum_{r=1}^{k} T(r) + \sum_{r=k+1}^{n} T(n-r)\right] \leq n + \frac{2}{n}\left[\sum_{r=\lfloor n/2 \rfloor}^{n} T(r)\right]$$

Therefore, $E[T(n)] = O(n)$.

4 Randomized Max-Cut

Recall that a cut in a graph $G = (V, E)$ is defined as a subset $S \subset V$. Cuts in graphs are useful structures with applications in network flows, statistical physics, circuit design, complexity and approximation theory.

We denote a cut as $[S, \overline{S}]$ and assume that S is not a trivial cut, i.e. $S \neq \emptyset$ and $S \neq V$. As shown in Figure 8, an edge (u, v) is said to be crossing the cut $[S, \overline{S}]$, if $u \in S$ and $v \in \overline{S}$.

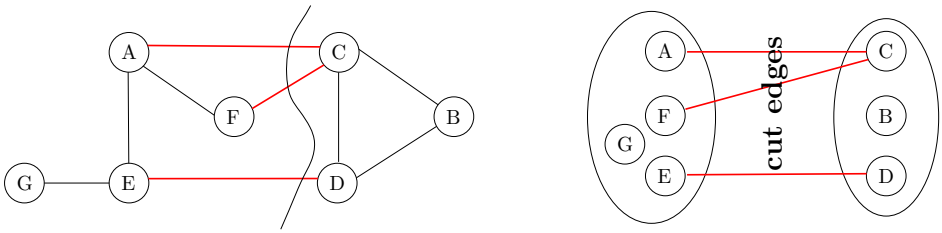


Figure 8: A cut $[\{A, F, E, G\}, \{C, B, D\}]$ in G

The size or cost of a cut is defined as the number of crossing edges. For weighted graphs, the size of cut is the sum of weights of crossing edges.

PROBLEM 2 (MAX-CUT(G) problem)**.** *: Find a cut in G of maximum size.*

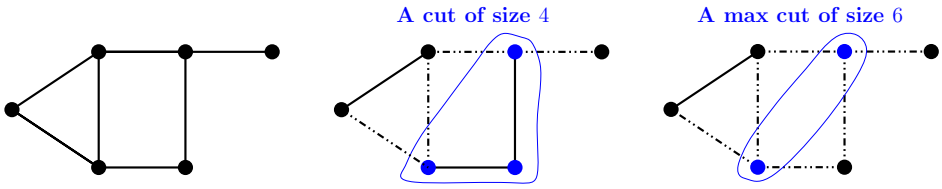


Figure 9: Examples of a cut and a max-cut in a graph

The decision version of the MAX-CUT(G) problem is NP-COMPLETE. A simple randomized approximation algorithm for MAX-CUT(G) is to place each vertex in S or $\overline{S}$ randomly, as shown in Algorithm 6.

Algorithm 6 RAND-MAX-CUT(G)

$S \leftarrow \emptyset$
for each vertex $v \in V$ **do**
  $S \leftarrow S \cup \{v\}$ with probability $1/2$

The runtime of the above algorithm is clearly $O(n)$ where $n = |V|$, since it requires only one iteration over the vertex set V. However, how 'good' is the estimated max-cut? We analyze it as follows.

For edge e, let C_e be an indicator random variable, where $C_e = 1$ if e crosses the cut $[S, \overline{S}]$ and $C_e = 0$ otherwise. Then, the for a given cut, $size([S, \overline{S}]) = \sum_{e \in E} C_e$. By linearity of expectation, $E\left[size([S, \overline{S}])\right] = E\left[\sum_{e \in E} C_e\right] = \sum_{e \in E} E[C_e]$.

$E[C_e]$ is the probability that e is a crossing edge in the cut. As shown in Figure 10, note that the $Pr[e = (u, v)$ crosses the cut $[S, \overline{S}]] = Pr\left[(u \in S \wedge v \in \overline{S}) \vee (u \in \overline{S} \wedge v \in S)\right] = 1/4 + 1/4 = 1/2$.

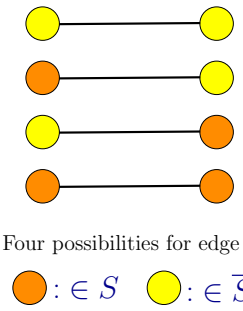


Figure 10: Given a cut $[S, \overline{S}]$ in $G = (V, E)$, each edge $e \in E$ must be one of these four types.

$$E\left[size([S, \overline{S}])\right] = \sum_{e \in E} E[C_e] = \sum_{e \in E} 1/2 = |E|/2 = m/2$$

Therefore, since the size of all cuts, including max-cut, is $\leq m$, where $m = |E|$, the cut constructed by this randomized algorithm has size at least $1/2$ of the max cut.

Although the algorithm is good on average, their is no (probabilistic) guarantee on the quality of its output. We can improve over RAND-MAX-CUT(G) using the magic of repeated trial to amplify the probability, i.e. run RAND-MAX-CUT(G) k times on G and return the largest cut found. We refer to this as a meta-algorithm of k repeated trails of RAND-MAX-CUT. The runtime for the meta-algorithm is $O(k(m+n))$ where building the cut takes $O(n)$ and determining its size takes $O(m)$.

For large k, the meta-algorithm returns a large cut with a high probability, which is the kind of guarantee we desire. In fact, we will show that we get a cut of size $m/4$ with very high probability.

Let $X_1, X_2, \cdots X_k$ be sizes of the cuts found by each run of RAND-MAX-CUT(G), and let $\mathcal{E}$ be the event when meta-algorithm of k repeated trials produces a cut of size less than $m/4$, i.e. $\mathcal{E} = \bigcap_{i=1}^{k} \left(X_i \leq \frac{m}{4}\right)$

Since X_i's are independent, by linearity of expectation,

$$Pr[\mathcal{E}] = Pr\left[\bigcap_{i=1}^{k}\left(X_i \leq \frac{m}{4}\right)\right] = \prod_{i=1}^{k} Pr\left[X_i \leq \frac{m}{4}\right]$$

To find $Pr\left[X_i \leq \frac{m}{4}\right]$, we will use the Markov Inequality.

Theorem 1. Markov Inequality: *If Z is a non-negative random variable and $a > 0$, then* $Pr[Z \geq a] \leq E[Z]/a \iff Pr[Z \geq aE[Z]] \leq 1/a$

Note that we can write $(X_i \leq \frac{m}{4})$ as $(m - X_i \geq \frac{3m}{4})$ to match the format of the Markov Inequality. For simplicity, let $Y_1, Y_2, \cdots Y_k$ be random variables, defined as $Y_i = m - X_i$. Since $m/2$ is a lower bound on the size of cut found by a single run X_i, $E[X_i] = m/2$. Thus, $E[Y_i] = m - E[X_i] = m - m/2 = m/2$. Then,

$$Pr[\mathcal{E}] = \prod_{i=1}^{k} P\left(Y_i \geq \frac{3m}{4}\right) \leq \prod_{i=1}^{k} \frac{E[Y_i]}{3m/4} = \prod_{i=1}^{k} \frac{m/2}{3m/4} = \prod_{i=1}^{k} \frac{2}{3} = \left(\frac{2}{3}\right)^k$$

If we output the largest cut out of k runs of RAND-MAX-CUT(G) k, the probability that we don't get $\geq m/4$ edges is at most $(2/3)^k$, i.e. the probability we do get $\geq m/4$ edges is at least $1 - (2/3)^k$. If we set $k = \log_{2/3} m$, then the probability we get at least $m/4$ edges is $1 - 1/m$.

Therefore, the meta algorithm is a $O((m+n)\log m)$-time randomized algorithm that finds a 0.25-approximation to $textscmax - cut$ with probability $1 - 1/m$. This is called a (ϵ, δ)-randomized approximation algorithm, but more on that later.

5 Minimum Cut

Similar to the MAX-CUT problem discussed above, we may also need to find a cut with minimum cost in a graph.

PROBLEM 3 (MIN-CUT(G) problem)**.** *: Find a cut in G of minimum size.*

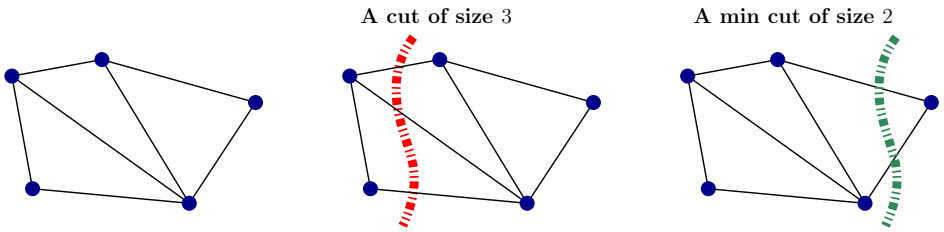


Figure 11: Examples of a cut and a min-cut in a graph

Note that the size of the minimum cut is at most the minimum degree of any vertex in G and as shown in Figure 12, a minimum cut does not have to be unique.

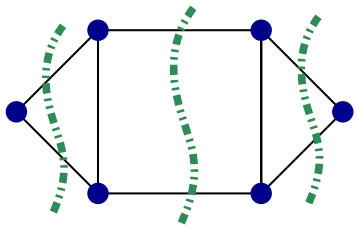
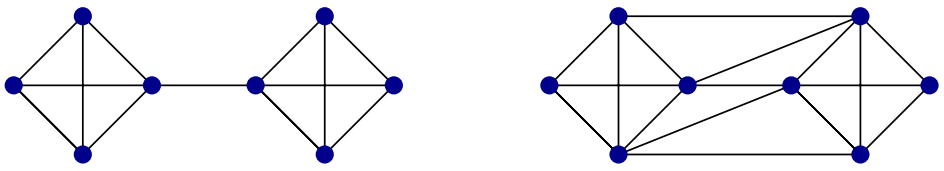


Figure 12: 3 minimum cuts, each of size 2

The problem is also called Global Min-Cut and has applications in network reliability and robustness analysis, as shown in Figure 13.



The network on the left is easier to disconnect

Figure 13: Knowing a min-cut allows an adversary to disrupt a network with minimal effort

Normalized minimum cut is used when spectral clustering is applied to image segmentation, in which a digital image is broken down into various subgroups. Suppose we want to separate the background of the image from the foreground to, for example, distinguish an aircraft or missile from the horizon. Figure 14 shows this separation for an example image and the application of Min-Cut to segmentation of this image is briefly described in Fig 15.

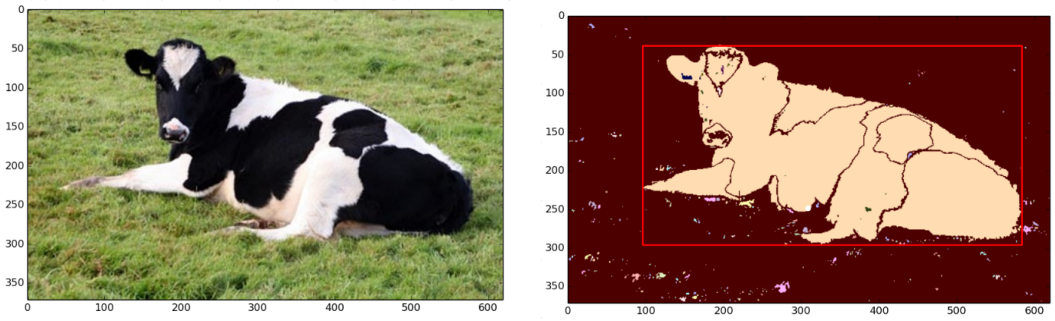


Figure 14: The original image (left) and its segmentation by a boundary separating the foreground from the background of the image (right)

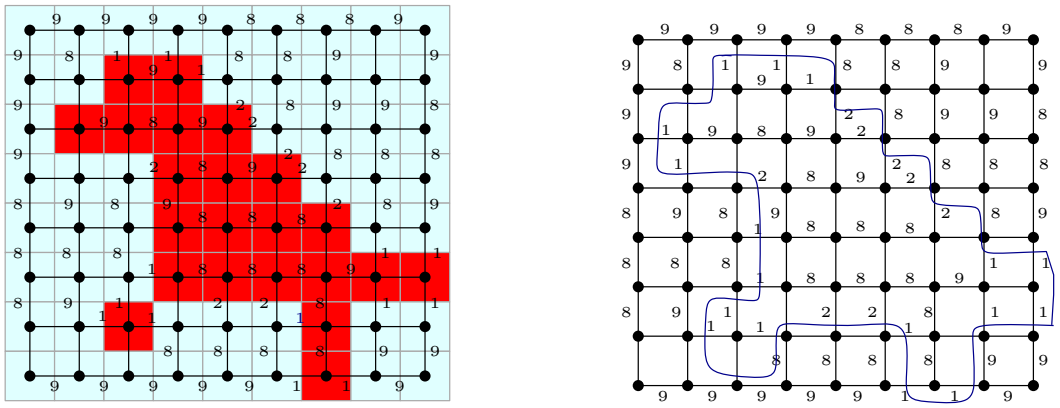


Figure 15: Image Segmentation using Min-Cut: Let each square denote a pixel. Red pixels belong to the foreground whereas white pixels denote the background. Observe that if pixel (x, y) is background or foreground, then so are nearby pixels. We make a graph (left) with nodes for each pixel adjacent to neighboring pixels and the weight of edge (i, j) is defined as a penalty p_{ij} of classifying i and j differently, i.e. p_{ij} is a 'similarity measure' determined by image processing. A min-cut in this weighted graph draws the boundary between the foreground and the background as it (overall) cuts across edges between nodes with least similarity.

Recall from network flows that a maximum $s-t$ flow in G is equal to a minimum $s-t$ cut. Since the value of the min-cut of G is minimum over all possible $s-t$ cuts in G, the brute force solution is to compute a min $s-t$ cut for all $s-t$ pairs of V. This requires $O(n^2)$ calls to a min $s-t$ cut (i.e.max $s-t$ flow) solver such as

- FORD-FULKERSON algorithm: $O(n^2 \cdot m \cdot |f_{max}|)$
- EDMOND-KARP algorithm: $O(n^2 \cdot n \cdot m^2)$
- DINIC's or push-relabel algorithm: $O(n^2 \cdot n^2 \cdot m)$

A smarter approach is to fix a node s which must appear in one of S or $\overline{S}$. and find the min $s-t$ cut for all $t \in V$. This requires only $O(n)$ calls to min $s-t$ cut (max $s-t$ flow) solver.

Many deterministic algorithms have been proposed for the MIN-CUT problem such as the Stoer-Wagner algorithm which takes $O(nm+n^2 \log m)$ time. We will study a simple randomized algorithm by Karger and an elegant extension of it by to Karger and Stein. These algorithms are based on the edge contraction operation. Before we discuss what the edge contraction operation is, recall the concepts of pseudographs and multigraphs, as shown in Figure 16.

Figure 16: A graph $G = (V, E)$, where V is the vertex set, is a pseudograph if E is a set of edges where G is a multigraph if E is a *multi-set of edges* i.e. there may be multiple edges between a pair of nodes. Both types of graphs may have self loops on nodes.

5.1 Edge Contraction

Contraction of an edge (u, v) in G constructs a graph $G \backslash \{u, v\}$ in which u and v become one vertex uv, edge (u, v) becomes a self-loop (which we remove in this case) and all edges incident on u or v become incident on uv. As shown in Figure 17, the resulting graph may become multigraphs, and we keep all edges introduced in the contraction process.

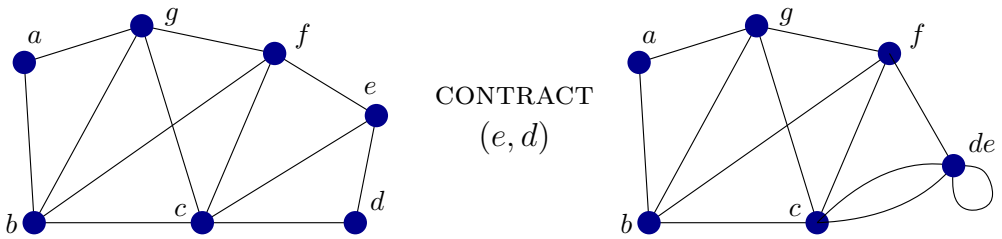


Figure 17: Example of edge contraction: the function CONTRACT(G, e) takes a graph and edge as input and return the resulting graph after contracting the given edge. A self loop introduced by the contraction is only shown in the first graph after contraction.

Edge contraction can be performed in $O(n)$ time bu merging the adjacency lists of u and v and updating the adjacency lists of neighbouring vertices can be updated in $O(n)$ time if we keep corresponding pointers at entries of adjacency lists (see Figure 18). We remove any self-loops in the resulting graphs. Multigraphs can be saved with multiplicity as edge weight.

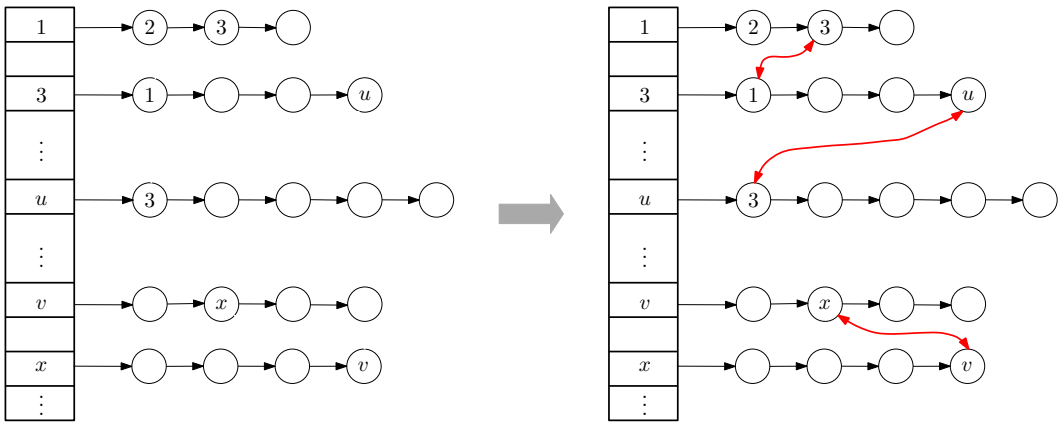


Figure 18: In each entry of the adjacency list, we maintain a pointer to the corresponding entry in the adjacency list of the other end-point. These pointers can be initially populated in one scan over the graph.

Now, how does edge contraction affect the minimum cut in the graph? The min cut in $G \setminus \{u, v\}$ is at least as large as the min cut in G because any cut in $G \setminus \{u, v\}$ is 'actually' a cut in G too. However, the converse is not necessarily true, as shown in Figure 19.

Figure 19: The min cut in G is not necessarily at least as large as the min cut in $G \setminus \{u, v\}$. The min-cut in the graph on the left is $[\{a, b, c\}, \{e, f, g\}]$ is of size one, whereas there is no cut of size one in the graph after contracting edge (c, f) on the right.

5.2 Karger's Algorithm

Karger's algorithm for the estimating the minimum cut simply contracts a random edge in a graph until there are only two vertices or 'super nodes' left in the graph. The estimated min-cut is then the cut induced by the two super nodes. For example, if the last two vertices remaining are (uv) and (xyz), the cut is $[\{u, , v\}, \{x, y, z\}]$.

Algorithm 7 : Karger's algorithm for mincut (G)

while there are more than two vertices left in G **do**
  Pick a random edge $e = (u, v)$
  $G \leftarrow G \setminus uv$
return G ▷ the cut induced by the remaining two (super)nodes

The estimated cuts from two runs of Algorithm 7 are shown in Figure 20.

Since each contraction reduces the number of vertices by 1, the number of contraction done by Karger's algorithm is $n - 2$. With the right data structure, a contraction can be done in $O(n)$ so the total runtime is $O(n^2)$.

Let us now analyze Karger's algorithm for the 'goodness' of the estimated min-cut. Let $C = [S, \overline{S}]$ be a cut. The intuition is that, if during the execution some edge in C is contracted, the algorithm will not output the cut C because if $(u, v) \in C \leftrightarrow u \in S \wedge v \in \overline{S}$ is contracted, then u and v will belong to the same supernode and (u, v) cannot be crossing edge. Since edges are selected at random for contraction, a cut with more number of crossing edges is more likely to not be output by the algorithm. The algorithm will output C only if it never contracts any edge in C. Among all cuts, min-cuts have the least probability of having an edge contracted. However, this intuition is alone not enough and we must obtain a probabilistic guarantee on the likelihood of a min-cut being output by the algorithm.

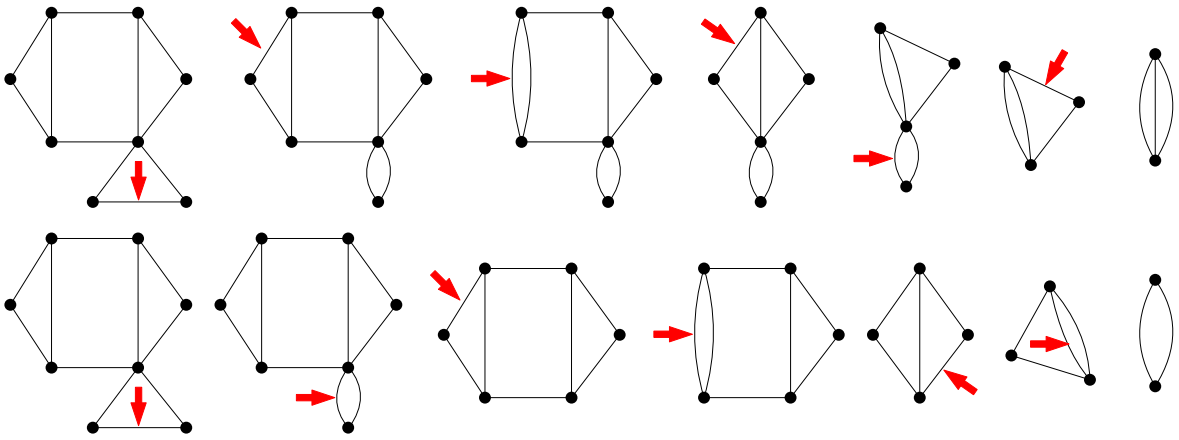


Figure 20: Two runs of the Karger's algorithm (the arrow marks the contracted edge in each step) producing a sub-optimal cut of size 3 (top) and an optimal min-cut of size 2 (bottom).

Let$G_i = (V_i, E_i)$ be the graph after the i^{th} contraction for $0 \leq i \leq n-2$, which means that $|V_i| = n_i = n - i$. Note that initially, $G_0 = (V_0, E_0) = G = (V, E)$ with $|V_0| = n$ and $|E_0| = m$.

Let $C = [S, \overline{S}]$ be a specific min-cut of size k. Since C is a min-cut of size k, every vertex in V has degree $\geq k$. Combining this information with the hand-shaking lemma (i.e. the sum of degrees of vertices equals twice the number of edges in the graph) we get that $m_0 \geq {}^{kn_0}/_2$ and more generally, $m_i \geq {}^{kn_i}/_2 = {}^{k(n-1)}/_2$.

We say that an edge is 'killed' if it is contracted in a given 'round' or iteration of the algorithm. We compute the probability that the min-cut C of size k is the output of the algorithm as follows.

$Pr[C$ is "killed" in 1st round$] = Pr[$an edge in C is contracted$] = {}^{k}/_{m_0} \geq {}^{2}/_{n}$

$Pr[C$ survives in 1st round$] = Pr[$no edge in C is contracted$] \geq 1 - {}^{2}/_{n_0}$

$Pr[C$ sruvives in $(i+1)$th round $\mid C$ survived so far$] = 1 - {}^{k}/_{m_i} \geq 1 - {}^{2}/_{n-i}$

$Pr[C$ sruvives all rounds$] = \prod_{i=0}^{n-3} Pr[C$ sruvives round $i+1 \mid C$ survived so far$]$

$Pr[C$ sruvives all rounds$] = Pr[C$ is the output$] = \prod_{i=0}^{n-3} \frac{n-i-2}{n-i}$

$Pr[C$ is the output$] = \frac{n-2}{n} \times \frac{n-3}{n-1} \times \frac{n-4}{n-2} \times \ldots \times \frac{2}{4} \times \frac{1}{3} = \frac{2}{n(n-1)} = {}^{1}/{\binom{n}{2}}$

Therefore, the probability that a min-cut is output by the Karger's algorithm is approximately${}^{1}/_{n^2}$, which seems to be very small. However, there are 2^m cuts, many of which are min-cuts, and the algorithm finds one of the min-cuts with probability ${}^{1}/_{n^2}$.

Nevertheless, with repeated trials, we can amplify the probability to any desired value, i.e. run the Karger's algorithm multiple times, say M times, and return the smallest of the M cuts obtained. Suppose we call this the GOOD-MIN-CUT(G, M) algorithm. Then,

$Pr[\text{all } M \text{ runs fail to output } C] \;=\; \prod_{i=1}^{M} Pr[\text{Run } i \text{ fails}] \;\leq\; \left(1 - 1/n^2\right)^M$.

A very useful inequality that we will use now is: $\forall\, x \in \mathbb{R}\;\; (1 + x) < e^x$. Applying this inequality to the probability that all GOOD-MIN-CUT(G, M) fails to output C, we get that:

$Pr[\text{GOOD-MIN-CUT}(G, M) \text{ fails to output } C] \;\leq\; e^{M/n^2}$.

If we set $M = cn^2 \log n$, $Pr[\text{GOOD-MIN-CUT}(G, M) \text{ outputs } C] \;\geq\; 1 - 1/n^c$.

In this case when $M = cn^2 \log n$, the runtime of GOOD-MIN-CUT is $O(n^4 \log n)$.

5.3 Karger-Stein Algorithm

In the GOOD-MIN-CUT algorithm, we repeat all $n-2$ rounds of Karger's algorithm each time. The Karger-Stein extension of the algorithm, however, tries to do this smartly. Let's analyze the probability of a min-cut C being 'killed' (i.e. an edge in C being contracted) in each round.

$Pr[C \text{ is "killed" in round 1}] = Pr[\text{an edge in } C \text{ is contracted}] = k/m_0 \geq 2/n$

$Pr[C \text{ is "killed" in round 2} \mid C \text{ survived round 1}] = k/m_1 \geq 2/n-1$

$Pr[C \text{ is "killed" in rond } (i+1) \mid C \text{ survived so far}] = k/m_i \leq 2/n-i$

$Pr[C \text{ is "killed" in rond } (n-3) \mid C \text{ survived so far}] \leq 2/4$

$Pr[C \text{ is "killed" in rond } (n-2) \mid C \text{ survived so far}] \leq 2/3$

As you can see, the probability of wrong contraction increases in each round. The intuition is that we can repeat only the rounds where the probability of C being 'killed' is higher to make it more likely that we can come across a round where it is not 'killed' and not waste time repeating the first 'few' iterations. In other words, as G gets smaller, repeat increasingly many times to reduce the error probability. This is, in-fact, what the Karger-Stein algorithm does.

As outlined in Algorithm 8, the Karger-Stein algroithm obtains two independently and randomly contracted graphs H_1 and H_2 from G. When H_1 and H_2 are small, it make 4 random contractions and so on. When the graph has less than 6 vertices, the minimum of all $\sim 2^5$ cuts is found. Note that Now we cannot chase a fixed minimum cut C, as both X_1 and X_2 could be min cuts, if successful and we may choose either.

Algorithm 8 : Karger-Stein algorithm for min-cut

```
1: function CONTRACT(G, t)
2:     while more than t vertices left in G do
3:         Pick a random edge e = (u, v)
4:         G ← G \ uv
5:     return G
6: function FAST-CUT(G)
7:     if n ≤ 6 then
8:         return exact min-cut via brute force
9:     t ← ⌈1 + n/√2⌉
10:    H_1 ← CONTRACT(G, t)
11:    H_2 ← CONTRACT(G, t)
12:    C_1 ← FAST-CUT(H_1)
13:    C_2 ← FAST-CUT(H_2)
14:    return smaller of C_1 and C_2
```

Let $T(n)$ be runtime of FAST-CUT(G) with $|V(G)| = n$.

$$T(n) = \begin{cases} 2T\left(n/\sqrt{2}\right) + O(n^2) & \text{if } n > 6 \\ O(1) & \text{else} \end{cases}$$

Then, by the master theorem, $T(n) = O(n^2 \log n)$, which is not much worse than the $O(n^2)$ of the initial Karger's algorithm.

Finally, we analyze the quality of the Karger-Stein algorithm.

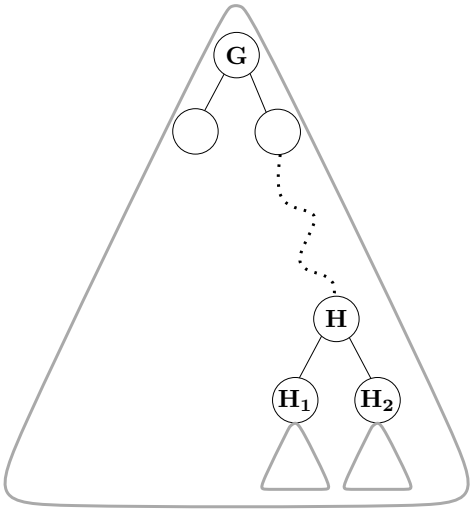
FAST-CUT(G) succeeds (i.e. finds an optimal min-cut) if and only if (a) a min-cut survives the CONTRACT(G, t) step and (b) at least one of the FAST-CUT(H_1) and FAST-CUT(H_2) finds a min-cut.

Probability a min cut survive CONTRACT(G, t) step (lines 10 and 11 in Algorithm 8) is:

$$Pr[\text{a cut survives } n-t \text{ contractions}] = \prod_{i=0}^{n-t-1} \frac{n-i-2}{n-i}$$
$$= \frac{n-2}{n} \times \ldots \times \frac{t}{t+2} \times \frac{t-1}{t+1} = \frac{t(t-1)}{n(n-1)}$$
$$= \frac{t(t-1)}{n(n-1)} \simeq \frac{1}{2} \text{ as } t = \frac{n}{\sqrt{2}}$$

Let $P(j)$ be the probability that FAST-CUT(H) finds min-cut if $|V(H)| = j$. FAST-CUT(G) finds a min-cut when either of the two following branches are successful:

- A min-cut survives in H_1 **AND** C_1 is a min-cut in H_1
- A min-cut survives in H_2 **AND** C_1 is a min-cut in H_2



The probability that branch i succeeds (i.e. min-cut survives in H_i AND C_i is a min-cut in H_i) is $1/2P(t)$. Then, the probability $P(n)$ that FAST-CUT(G) succeeds is the probability that both branches do not fail. Therefore, it can be easily proved by induction that

$$P(n) \geq 1 - (1 - 1/2P(t))^2 = 1 - (1 - 1/2P(n/\sqrt{2}))^2 = \Omega(1/\log n)$$

Therefore, the extended algorithm has a success probability $\Omega(1/\log n)$ which is much better than $\Omega(1/n^2)$ of initial version. Furthermore, to achieve success probability $\Omega(1 - 1/n^c)$, the initial version had to amplified by $cn^2 \log n$ independent trial with total

$O(n^4 \log n)$ runtime whereas the FAST-CUT(G) algorithm needs to be amplified by $c \log^2 n$ independent trials with total runtime $O(n^2 \log^3 n)$. Further improvements in these algorithms have been made, but are out of the scope of this course.

6 Max-3-SAT

Recall the 3-SAT(f) problem: Given n Boolean variables $x_1, \ldots, x_n$, where x_i can take a value of 0 or 1, a literal is a variable appearing in some formula as x_i or $\bar{x_i}$ and a clause of size 3 is an OR of three literals. A 3-CNF formula, which is AND of one or more clauses of size ≤ 3, is satisfiable if there is an assignment of 0/1 values to the variables such that the formula evaluates to 1 (or True). The 3-SAT(f) search problem is to find a satisfying assignment for the 3-CNF formula f. This problem is NP-HARD. An NP-HARD optimization problem in the same setting is MAX-3-SAT.

PROBLEM 4 (MAX-3-SAT(f) problem)**.** *: Find an assignment for* 3-CNF *formula f that satisfies the maximum number of clauses.*

The brute force algorithm for MAX-3-SAT(f) is to try all 2^n possible assignments in $\mathcal{O}(m2^n)$ time, which is infeasible. We study a more efficient randomized algorithm.

The key idea is very simple: toss a coin and set a variable to True if Heads and to False if Tails, i.e. independently set each variable true with probability $1/2$. The important question now is, what is the expected number of clauses satisfied by such a random assignment?

Theorem 2. *A random assignment to variables satisfies in expectation $7m/8$ clauses of a* 3-CNF *formula f with m clauses*

Proof. Let random variable $Z_j = 1$ if clause c_j is satisfied and $Z_j = 0$ otherwise. Then, $E[Z_j] = Pr[C_j \text{ is satisfied}] = 1 - Pr[C_j \text{ is not satisfied}]$. Evidently, C_j is not satisfied when all literals in C_j are set to FALSE. Since literals values are set independently, $Pr[C_j \text{ is not satisfied}] = (1/2)^3 = 1/8$. Thus, $E[Z_j] = 7/8$.

Let Z be the number of clauses satisfied by random assignment. By linearity of expectation, $E[Z] = \sum_{j=1}^{m} E[Z_j] = \sum_{j=1}^{m} \frac{7}{8} = \frac{7m}{8}$. □

Theorem 3. *For any instance of* MAX-3-SAT *with m clauses, there exists a truth assignment which satisfies at least $7m/8$ clauses.*

Proof. There is a non-zero probability that a random variable takes the value of its expectation, i.e. $Pr[Z \geq E[Z] = 7k/8] > 0$. This non-constructive way by Paul

Erdös of proving a claim is called Probabilistic Method which refers to proving the existence of a non-obvious property by showing that a random construction produces it with positive probability. □

Although there exists a truth assignment which satisfies at least $7m/8$ clauses, we have no probabilistic guarantees for the above randomized algorithm. Again, by utilizing the standard trick of repeated trials, we can repeatedly generate a random assignment A to variables until A satisfies at least $7m/8$ clauses. But how long will that take? Can we guarantee that this can be achieved in polynomial time? Turns out, yes we can, in expectation at least. In fact, this gives us a $7/8$ Las Vegas approximation algorithm for MAX-3-SAT, i.e. an algorithm guaranteed to find an assignment satisfying at least $7m/8$ clauses whose expected runtime is polynomial.

In what case would the runtime be polynomial? Suppose $Pr\left[A \text{ satisfies} \geq 7m/8 \text{ clauses}\right] \geq p$. Then, by expectation of a geometric random variable, expected number of trials to find this assignment is $1/p$. If p is polynomial, then expected running time is polynomial. Therefore, we show that p is polynomial.

Theorem 4. *The probability p that a random assignment satisfies $\geq 7m/8$ clauses is $\geq 1/8m$*

Proof. A lower bound on p can be obtained by using $E[Z] = 7m/8$ where Z is the number of clauses satisfied by a random assignment. Let p_j be the probability that the random assignment satisfies exactly j clauses, where $j = 1, 2, \cdots, m$. Then,

$$E[Z] = \sum_{j=0}^{m} j\, p_j = \sum_{j<7m/8} j\, p_j + \sum_{j\geq 7m/8} j\, p_j \leq 7m-1/8 \sum_{j<7m/8} p_j + m \sum_{j\geq 7m/8} p_j$$

This implies that $E[Z] \leq 7m-1/8 \cdot 1 + m \cdot p$ which in turn implies that $7m/8 \leq 7m-1/8 + mp$. Solving this inequality for p shows that $p \geq 1/8m$. □

In fact, it was proven by Hástad in 1997 that this lower bound is tight, i.e. MAX-3-SAT cannot be approximated in polynomial time to within a ratio greater than $7/8$, unless P = NP.

6.1 Derandomization

So far, in all the randomized algorithms that we have seen, random choices made by an algorithm sometimes happen to be 'good' when the algorithm's output is close to the optimal. Is it possible that we can always make these 'good' choices, i.e. can

these be made deterministically? For some problems and algorithm, yes, it is possible. This process of transforming a randomized algorithm into a deterministic algorithm is called *derandomization*. We show how the 7/8 Las Vegas algorithm for MAX-3-SAT can be derandomized.

In order to derandomize the 7/8 Las Vegas algorithm for MAX-3-SAT, we need to know which set of choices for variable assignments are 'good', i.e. satisfies a greater number of clauses. The idea is to consider the choice for each variable, either True or False, one at a time. Given assignments for the "first i" variables $x_1 = a_1 \cdots, x_i = a_i$, the expected value of the number of satisfied clauses with random assignment of the unassigned variables $x_{i+1}, \cdots, x_n$ can be computed in polynomial time. For each clause C_j when a variable is given an assignment, if the corresponding literal in the clause evaluates to False, then remove the literal from C_j. Otherwise, if the corresponding literal in the clause evaluates to True, then the clause can be ignored for remaining variable assignments as it is already satisfied. This yields the following polynomial time deterministic algorithm for MAX-3-SAT.

First, fix an order of variables $x_1, x_2, \cdots, x_n$. Traverse the variables in the fixed order, setting the value of the next variable in each iteration. How is the value of x_i set? x_i should be set to the value given which, the conditional expectation of the number of satisfied clauses is greater, i.e. if $E[Z|x_1 = a_1, \cdots, x_{i-1} = a_{i-1}, x_i = \text{TRUE}] > E[Z|x_1 = a_1, \cdots, x_{i-1} = a_{i-1}, x_i = \text{FALSE}]$, then $x_i = \text{TRUE}$ and otherwise $x_i = \text{FALSE}$. How is the conditional expectation computed? Let Z be the number of clauses satisfied. The conditional expectation of Z is the unconditional expectation of Z in the reduced set of clauses plus the number of already satisfied clauses.

Since $E[Z|x_1 = a_1, \cdots, x_i = a_i] \geq E[Z]$ for $1 \leq i \leq n$ by construction and $E[Z] = 7m/8$, it follows that $E[Z|x_1 = a_1, \cdots, x_i = a_i] \geq 7m/8$. Therefore, the derandomized algorithm for MAX-3-SAT deterministically satisfies at least $7m/8$ clauses.

7 Closest Pair

For two points $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$, the Euclidean distance defined as $d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ which can be computer in $O(1)$.

PROBLEM 5 (CLOSEST-PAIR(P) problem:). *Given a set $P = \{p_1, p_2, \ldots, p_n\}$ of n distinct points in $\mathbb{R}^2$, find a pair of distinct points (p_i, p_j) in P that minimizes $d(p_i, p_j)$*

The closest pair problem finds various applications in computer graphics, computer vision, geographic information systems, molecular modeling and air traffic control.

The naive brute force algorithm is to find the minimum distance among all $\binom{n}{2}$ pairwise

distances in $O(n^2)$ time. In case of 1-dimensional space, a straight-forward algorithm is to sort the points in $O(n \log n)$ time and find the closest adjacent points in $O(n)$. For 2-dimensional space, we have already seen a Divide and Conquer algorithm that takes $O(n \log n)$ time. Now we will study a randomized algorithm for the CLOSEST-PAIR(P) problem whose expected runtime in $O(n)$.

We assume that the distance between each pair of points is distinct and that all points are in the unit square $0 \leq x_i, y_i \leq 1$ without loss of generality. $=$

Let $P = \{p_1, p_2, \cdots, p_n\}$ be a fixed random order of points and let $S_i = \{p_1, p_2, \cdots, p_i\}$ be the set of the first i points in P. We denote the distance of the closest pair in S_i δ_i by δ_i, i.e. the distance of the closest pair in P is δ_n. The idea is to begin with S_2 laid out a grid G with cell size $\delta_2 \times \delta_2$. In each following iteration, we add the next random point p_i to the grid and update the cell size of the grid to δ_i if $\delta_i < \delta i - 1$, as shown in Figure 21.

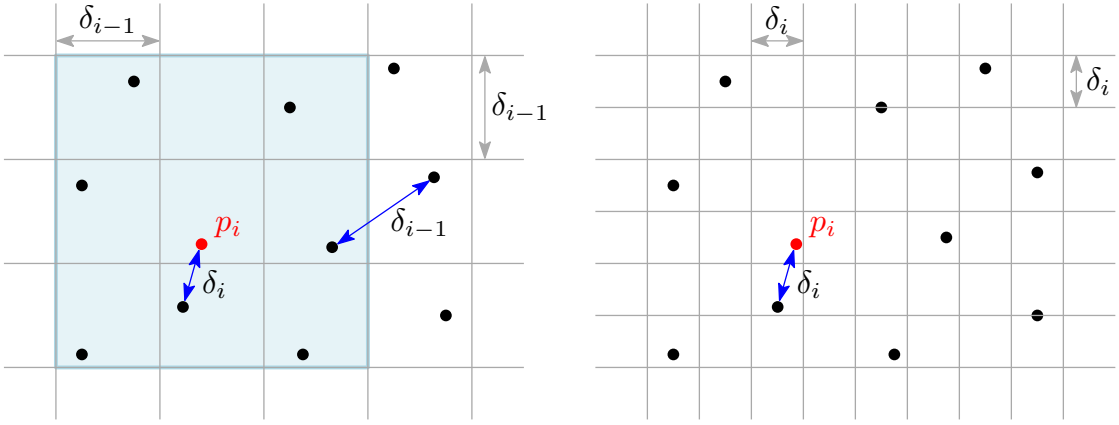


Figure 21: Point p_i added to the grid (left). The highlighted cells constitute the neighbourhood $N(p_i)$ of the cell containing p_i which is used to compute δ_i efficiently. Grid cell size is updated as $\delta_i < \delta i - 1$(right).

A naive method to compute δ_i is to compute the distance of p_i with all points in S_{i-1}, which will result in overall quadratic time as we would have eventually have compared all pairwise distances. An efficient way to compute δ_i in constant time is to utilize the fact that the distance between p_i and points outside the adjacent cells of the cell containing p_i, i.e. outside the neighbourhood $N(p_i)$ as shown in Figure 21, is already at least δ_{i-1} by construction since the cell size of the grid is the minimum pairwise distance computer so far. Therefore, it is sufficient to compare the minimum among the distances of p_i with the eight adjacent cells, where each cell contains at most one point by construction, with δ_{i-1}. The procedure is outline in Algorithm 9.

To implement the grid structure, we identify the following required operations:

1. BUILD-GRID(S, δ): build grid G with cell size δ & insert all points in S
2. INSERT-POINT(p_i): insert p_i
3. LOCATE-CELL(p_i): return cell containing p_i
4. GET-POINTS(c): return points in cell c

The grid can be implemented using hashing such that all operations take $\mathcal{O}(1)$ time. The key universe is IDs of all cells in the grid, whereas the actual key space is the IDs of cells containing points. The point co-ordinates are the data for each key. The cell containing p_i is located at the cell $(\lfloor x_i/\delta_{i-1} \rfloor, \lfloor y_i/\delta_{i-1} \rfloor)$ in the grid.

Algorithm 9 Randomized Closest Pair: returns distance

```
function CLOSEST-PAIR(P)
    {p_1, p_2, ..., p_n} ← RANDOM-PERMUTATION(P)
    S_2 ← {p_1, p_2}
    G ← BUILD-GRID(S, δ_2)
    for i = 3 → n do
        S_i ← S_{i-1} ∪ p_i                          ▷ O(1)
        Compute δ_i                                  ▷ O(1)
        if δ_i < δ_{i-1} then G.BUILD-GRID(S, δ_i)   ▷ O(i)
        else
            G.INSERT-POINT(p_i)                      ▷ O(1)
    return δ_n
```

Note that the runtime of the above algorithm depends on whether the grid is rebuilt with the updated cell size in an iteration Thus, to analyze the expected runtime, we first need to find the probability of the event $\delta_i < \delta_{i-1}$ occurring for a given permutation of S_i.

Let C_i be the closest pair in S_i. Given S_i, $\delta_i < \delta_{i-1}$ when $p_i \in C_i$ for any permutation of S_i. Thus $Pr[\delta_i < \delta_{i-1}|S_i] = \frac{2(i-1)!}{i!} = \frac{2}{i}$. Then, the unconditional probability of $\delta_i < \delta_{i-1}$ is:

$$Pr[\delta_i < \delta_{i-1}] = \sum_{j \in \binom{n}{i}} Pr[\delta_i < \delta_{i-1}|S_{i_j}] \cdot Pr[S_{i_j}]$$

Since all $\binom{n}{i}$ choices of S_i are equally likely, the probabilities of all permutations occurring add up to 1 i.e. $\sum_{j \in \binom{n}{i}} Pr[S_{i_j}] = 1$. Using this fact and applying linearity of expectation,

$$Pr[\delta_i < \delta_{i-1}] = \frac{2}{i} \sum_{j \in \binom{n}{i}} Pr[S_{i_j}] = \frac{2}{i}$$

Let the runtime of an iteration i be X_i and the overall runtime of the algorithm be X.

$E[X_i] = \mathcal{O}(1) + \mathcal{O}(i) \cdot Pr[\delta_i < \delta_{i-1}] = \mathcal{O}(1) + \mathcal{O}(i) \cdot 2/i = \mathcal{O}(1)$.

By linearity of expectation, $E[X] = \sum_{i=1}^{n} E[X_i] = \mathcal{O}(n)$.

Therefore, the expected runtime of the randomized algorithm for CLOSEST-PAIR(P) is linear.

8 Hashing

8.1 Dictionary ADT

From your Data Structures course, recall that the *dictionary* ADT (Abstract Data Type) maintains a *set* of n elements from a universe U. The elements are unique and identified by their 'key' k and could have a particular value or satellite data associated with the key, i.e. e;lements could be compound (key, value) pairs. For example, a dictionary may contain student IDs are key and their exam score as the corresponding value. Another dictionary keyed by student IDs may store each student's personal information such as age, gender, address, etc. The main operations required for the dictionary ADT are INSERT, LOOKUP and DELETE. A dictionary can be implemented in various ways: using arrays and linked lists which may be sorted or unsorted, binary search trees which may be balanced or unbalanced, and hash tables. Table 1 summarizes the time complexity of the key operations for implementation using each of these data structures.

Operation	Unsor. Array	Sorted Array	Unsorted Linked list	Sorted Linked list	BST	AVL	Hash Function
Search(D,k)	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(h)$	$O(\log n)$	$O(1)$
Insert(D,k,v)	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(h)$	$O(\log n)$	$O(1)$
Delete(D,k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(h)$	$O(\log n)$	$O(1)$

Table 1: Runtime of dictionary operations for different implementations

In many applications, it is vital to perform all three operations efficiently in constant time. A simple straight-forward method to ensure all operations are done in $O(1)$ is to use a direct-address table, as shown in Figure 22, where each position in the table corresponds to a key in the universe U.

Image: CLRS

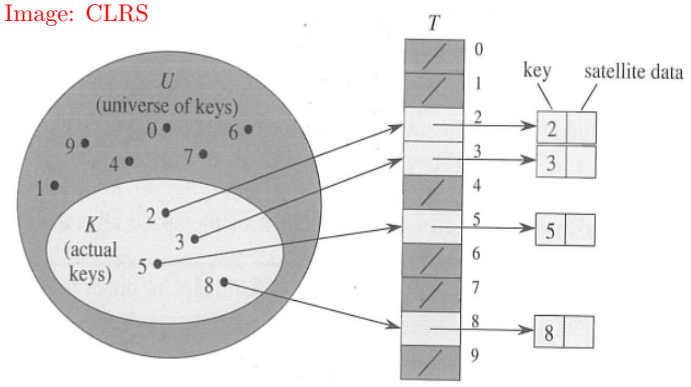


Figure 22: Dictionary implementation using a direct-address table

An evident drawback of this implementation is the large unused space occupied as a result of a large universe. The space used can be reduced if there is no satellite data since then, keys can be stored in a bit-vector and for a fixed number of elements, the space taken by a bit-vector is much smaller than that taken by an array. However, there is still a large percentage of unused space that is occupied. To see how $O(1)$ time complexity can be achieved without wasting space, we introduce hash tables.

8.2 Chained Hash Tables

Let $m \in \mathbb{Z}^+$ be a positive integer and let $h : U \to [m]$ be a function that maps elements in U to integers $[1, m]$. The hash table T, then, is an array (or table) of length m. For an element x with key k, store at $T[h(k)]$ to INSERT x, return or get $T[h(k)]$ to INSERT

x and remove from $T[h(k)]$ to DELETE x. Since $T[h(k)]$ can be computed in constant time, all operations take $O(1)$.

Image: CLRS

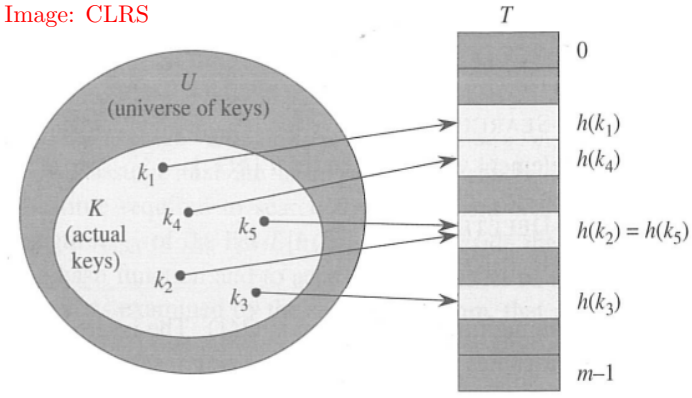


Figure 23: Dictionary implementation using a hash table which results in a collision as elements k_5 and k_2 map to the same index in the table.

As shown in Figure 23, a collision occurs if $h(k_x) = h(k_y)$ and it is unclear which of k_x or k_y should be stored. Most likely, we want to store both k_x and k_y, for which we used chained hash tables.

Instead of storing a single element at each position in the table T, we let $T[i]$ be an array or list for $i \in [1, m]$. The operations are carried out similar to hash tables but the element x with key k is now inserted, looked up and deleted in the list $T[h(k)]$ instead of $T[h(k)]$ itself, as shown in Figure 24.

Image: CLRS

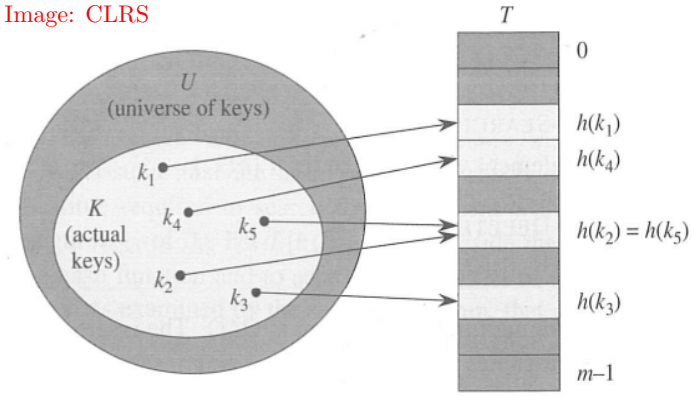


Figure 24: Dictionary implementation using a chained hash table.

The runtime of all operations for data x with key k now depends on the length of the list at $T[h(k)]$. It would be a reasonably efficient algorithm if we can somehow ensure

that the length of lists in T is not too large. As it turns out, we can obtain probabilistic guarantees by introducing randomization in the hashing (mapping elements in U to lists in T using h) process.

8.3 Randomized Hashing

Note that since an element must always hash (or be mapped) to the same list in T for correct storage and retrieval, the hash function h can not involve randomness. However, given multiple hash functions, we can randomly choose which one to use for each element, i.e. for $z \in U$, $h(z)$ is chosen uniformly at random from $\{0, \cdots, m-1\}$. We now analyze the expected runtime of operations in chained hash tables using randomized hashing.

For any $x_i \in U$, let random variable $C_i = 1$ if $h(x_i) = h(z)$ and $C_i = 0$ otherwise. Let X be the number of elements in the same list as z, i.e. $X = \sum_{x_i \neq z} C_i$. Then, by linearity of expectation,

$$E[X] = E[\sum_{x_i \neq z} C_i] = \sum_{x_i \neq z} E[C_i] = \sum_{x_i \neq z} Pr[h(x_i) = h(z)] = \sum_{x_i \neq z} \frac{1}{m} \leq \frac{n}{m}$$

Therefore, the expected runtime of operations is $\mathcal{O}(1 + E[X]) = \mathcal{O}(1 + {}^{n}/_{m})$. We observe a space-time tradeoff, i.e. larger m leads to a lower expected runtime since the more lists (or hash functions) are created, the shorter each list is expected to be for a fixed number of elements n. We desire a small range (m) of hash functions due to space limitations, and fewer collisions for efficient operations. Furthermore, since each operation requires evaluating a hash value, it must be easy to do so for any key with small space complexity.

We can obtain probabilistic guarantees is universal hash functions defined as follows:

Theorem 5. *A family of hash functions $\mathcal{H}$ is 2-universal iff for any $x, y \in_{x \neq y} U$, if $h \in \mathcal{H}$ is chosen uniformly at random, then $Pr[h(x) = h(y)] \leq {}^{1}/_{m}$*

An example of universal hash functions is the Linear Congruential Generators for $U = Z$. For a prime number $p > m$ and any two integers a and b such that $1 \leq a \leq p-1$ and $0 \leq b \leq p-1$, a hash function $h_{a,b} : U \mapsto [m]$ is defined as $h_{a,b}(x) = [(ax + b) \mod p] \mod m$. Then, $\mathcal{H} := \{h_{a,b} \ : \ 1 \leq a \leq p-1 \ , \ 0 \leq b \leq p-1\}$ is 2-universal. This makes picking a random $h \in \mathcal{H}$ easy since it amounts to selecting random a and b.

Another interesting example of hashing with probabilistic guarantees is Locality-Sensitive-Hashing (LSH) which is widely used in situations where algorithms work in a very limited space such as those required in the data streaming model were the data is

far larger than available memory. Although we do not dive into the LSH scheme, we discuss how randomization is crucial to processing of streaming data in the following section.

9 Stream Processing

Stream processing is the application of data analysis and algorithmic methods on a continuous data stream which is a massive sequence of data items too large to store on disk, memory, cache, etc. For example, data from social media (twitter feed, foursquare checkins), sensors (weather, radars, cameras, IoT and energy devices), network traffic (trajectories, source/destination pairs), financial and satellite feeds, sequences of web clicks and search queries is processed in a streaming manner. Before we see how this kind of data is dealt with, and where randomization lies in this context, we observe the key characteristics of data streams:

- Huge volumes of continuous data, possibly infinite
- Arbitrary arrival order of items
- Fast changing and requires fast, real-time response
- Captures aptly our data processing needs of today
- Random access is expensive
- Constant per item processing required
- Single pass or scan over the stream allowed in most cases
- Store only the summary of the data seen so far due to limited memory
- Most stream data are low-level or multidimensional in nature and thus need multi-level and multi-dimensional processing

Data items can be of complex types such as documents (tweets, news articles), images, geo-located time-series, etc. For simplicity, we can abstract away application specific details and study the basic algorithmic ideas considering the data stream as a sequence of numbers.

9.1 Stream Model of Computation

A data stream is a sequence of m items $\mathcal{S} = a_1, a_2, a_3, \ldots, a_m$, where each item is chosen from some universe of size n. Typically we take the universe to be $[n] := \{1, 2, \ldots, n\}$.

n and m are two size parameters. m may be unknown and we do not assume anything about the distribution of items.

Typically we work in the model where we see each item only once in the order given by $\mathcal{S}$, in general we cannot (or do not want to) save the whole stream. In the literature there are algorithms that take more than one passes over the stream but here we restrict ourselves to one-pass algorithms only.

Our goal is to use a very small amount of memory for computing some function over stream, $f(\mathcal{S})$, i.e. space requirement of algorithm should be $o(\min\{m, n\})$. Ideally we should use $O(\log n + \log m)$ space. Note that this space is needed anyway to store one (or some constant number of) stream items for processing. Sometimes space requirement could be relaxed to polylogarithmic in $\min\{m, n\}$ (like $(\log n)^c$ for some constant c).

In most of interesting cases of computing functions over streams, $f(\mathcal{S})$ can provably not be computed given the sublinear space and 1-pass requirements. We, therefore often allow approximation algorithms that make errors with bounded probability.

9.1.1 Synopsis

The fundamental methodology in stream processing is to keep a synopsis, i.e. a succinct summary of the stream that has arrive so far and answer query based on it. The synopsis is updated after examining each item in $O(1)$ and occupies space of the order of poly-log bits. There are various kinds of synopsis which can be maintained, such as sliding window, random sample, histogram, wavelets and sketch. Figure 25 outlines the stream processing model.

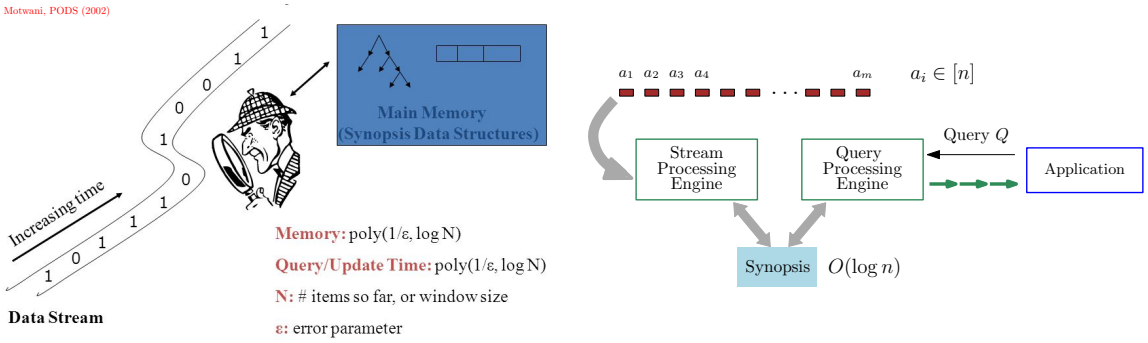


Figure 25: Data Stream Model of Computation

Using a small synposis, we can compute quite a few functions of the stream $\mathcal{S} = a_1, a_2, \ldots, a_m$, such that each $a_i \in [n]$. These examples are included as a motivation and to make the concepts of stream computation and synopsis clear.

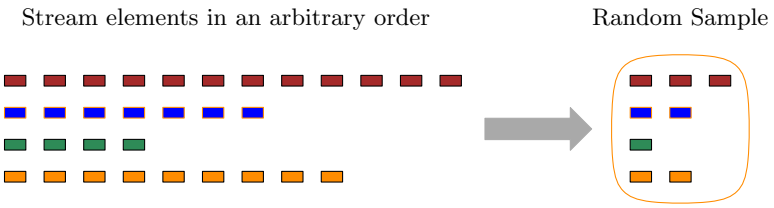


Figure 26: The random subset is a representative sample of the stream

- Length of $\mathcal{S}$ (m): This can clearly be computed by storing a running counter. The size of synopsis is one integer.
- Sum of $\mathcal{S}$: This can also be computed by storing a running sum. I.e. an integer variable initialized to 0 is added to the next stream item a_i. At the end of stream or at query time, the variable contain the sum of all elements of the $\mathcal{S}$.
- Mean of $\mathcal{S}$: The mean value can be computed from the sum and length of $\mathcal{S}$ (computed above). Note the size of synposis in this case is 2 integers.
- Variance of $\mathcal{S}$: This can be computed from keeping a synopsis of 3 integers to get the sum of elements of $\mathcal{S}$, sum of squares of elements of $\mathcal{S}$, and length of $\mathcal{S}$.

  $$Var(X) = E(X^2) - (E(X))^2$$

  Recall that computing variance using the definition of $Var(X) = E((X - \mu)^2)$ will not work in the streaming model.

All these use $O(\log n)$ bits memory and constant time per element. This is just to emphasize the fact, that though the streaming model is restrictive, we can still achieve quite a lot.

9.2 Random Sampling

A general and powerful technique to tackle massive data streams is *sampling*. Random sample is the most versatile general purpose synopsis with deep statistical foundations. Recall from your undergraduate statistics courses some relevant concepts like population, sample, confidence interval, size of sample, bias, weighted sampling, etc. We keep a "representative" subset of the stream as a synopsis and compute answers to the given query based on the sample (with appropriate scaling etc.) Probabilistic guarantees on the approximation quality of the query answer are derived using the fact that the sample is a (random) representative sample.

There are several sampling techniques to determine how a random sample of a data stream is kept. We first discuss the case when the entire data is given as an array or

list assuming the data is one-dimensional, and then see how one or more elements can be sampled from a stream.

PROBLEM 6. *Sample a random element from array A of length n. Generally, we require a uniform sample, i.e. each data point is chosen equally likely, (pick $A[i]$ with probability $1/n$).*

This problem can be solved as follows: Generate a random number $r \in [0, n]$. Many programming languages provide a pseudo random number generator function. Most of them generate real numbers in $[0, 1]$. Thus, to get a number in $[0, n]$ we can do $r \leftarrow \text{RAND}() \times n$. The number r is then rounded and we return $A[\lceil r \rceil]$.

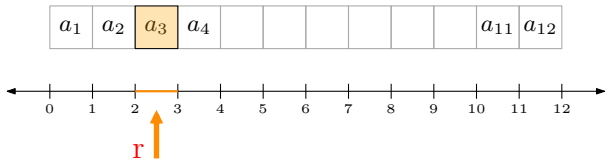


Figure 27: Sample an element from A (length 12)

9.2.1 Weighted Sampling

PROBLEM 7. *Given an array A of n elements, where the elements $A[i]$ has an associated weight w_i. Sample a random element (by weight) from A, i.e. choose $A[i]$ with probability w_i/W, where W is the total (sum of) weights of elements in A.*

Let $W_i = \sum_{j=1}^{n} w_i$ (this implies $W = W_n$), we generate a random number r in $[0, W]$. As discussed above this can be done for instance as $r \leftarrow \text{RAND}() \times W_n$. Then we return the element $A[i]$ such that $W_{i-1} \leq r < W_i$. This returns $A[i]$ with probability w_i/W because the probability that r is such that $W_{i-1} \leq r < W_i$ is $W_i - W_{i-1}/W = w_i/W$.

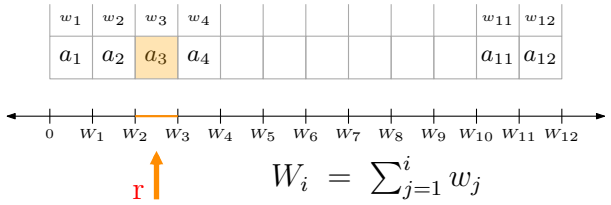


Figure 28: Sample random element (by weight) from array A of length 12

9.2.2 Reservoir Sampling

PROBLEM 8. *Given a stream $\mathcal{S} = a_1, a_2, \ldots,$. Sample a random element from S, i.e. choose the element a_i with probability $1/m$. Here m is the length of the stream.*

If m is known, we can use the algorithm for uniform sampling an element from an array. In other words, we pick a random index r between 1 and m and then choose a_r

(when it arrives). When m is unknown (think of it as follows: elements are streamed one at a time and at any time you may be asked to return a random sample. So m is known at the query time, but since we cannot store the whole stream, we have to be prepared with a random sample from the currently observed elements of the stream. The algorithm for this is called reservoir sampling.

Algorithm 10 : Reservoir Sampling ($\mathcal{S}$)

$R \leftarrow a_1$ ▷ R (reservoir) maintains the sample
for $i \geq 2$ **do**
  Pick a_i with probability $1/i$
  Replace with current element in R ▷ If a_i is picked

Let us analyze the probability that a given element a_i is chosen. The probability that a_i is in the reservoir R (at query time m or end of the stream) is given by

$$= \underbrace{\text{Pr that } a_i \text{ was selected at time } i}_{\frac{1}{i}} \times \underbrace{\text{Pr that } a_i \text{ survived in } R \text{ until time } m}_{\prod_{j=i+1}^{m}\left(1-\frac{1}{j}\right)}$$

$$= \frac{1}{\cancel{i}} \times \frac{\cancel{i}}{\cancel{i+1}} \times \frac{\cancel{i+1}}{\cancel{i+2}} \times \frac{\cancel{i+2}}{\cancel{i+3}} \times \ldots \times \frac{\cancel{m-2}}{\cancel{m-1}} \times \frac{\cancel{m-1}}{m} = \frac{1}{m}$$

PROBLEM 9. *Given a stream $\mathcal{S} = a_1, a_2, \ldots,$. Sample k random element from S, i.e. the element a_i should be in the sample with probability k/m. Again, m is the length of the stream.*

Reservoir sampling is easily extended to solve this problem.

Algorithm 11 : Reservoir Sampling ($\mathcal{S}, k$)

$R \leftarrow a_1, a_2, \ldots, a_k$ ▷ R (reservoir) maintains the sample
for $i \geq k+1$ **do**
  Pick a_i with probability k/i
  If a_i is picked, replace with it a randomly chosen element in R

The probability that a_i is in the reservoir R (at query time m or end of the stream) is given by

$$= \underbrace{\text{Pr that } a_i \text{ was selected at time } i}_{\frac{k}{i}} \times \underbrace{\text{Pr that } a_i \text{ survived in } R \text{ untill time } m}_{\prod_{j=i+1}^{m}\left(1-\left(\frac{k}{j}\times\frac{1}{k}\right)\right)}$$

$$= \frac{k}{\cancel{i}} \times \frac{\cancel{i}}{\cancel{i+1}} \times \frac{\cancel{i+1}}{\cancel{i+2}} \times \frac{\cancel{i+2}}{\cancel{i+3}} \times \ldots \times \frac{\cancel{m-2}}{\cancel{m-1}} \times \frac{\cancel{m-1}}{m} = \frac{k}{m}$$

$\mathcal{S}: a_1, a_2, a_3, a_4, \ldots, a_m$ $\quad a_i \in [n]$

$\mathbf{F}:$

1	2	3			n
f_1	f_2	f_3	$\cdots$	$\cdots$	f_n

$f_j = |\{a_i \in \mathcal{S} : a_i = j\}|$ (frequency of j in $\mathcal{S}$)

$\mathcal{S}: 2, 5, 6, 7, 8, 2, 1, 2, 7, 5, 5, 4, 2, 8, 8, 9, 5, 6, 4, 4, 2, 5, 5$

$\mathbf{F}:$

1	2	3	4	5	6	7	8	9
1	5	0	3	6	2	2	3	1

Figure 29: Example of a linear sketch

A random sample is a general purpose synopsis and can be used to answer any query about the whole stream. However, in sampling, we only process the sampled elements and do not take any advantage from observing the whole stream. Sketches, histograms and wavelets take advantage from the fact the processor see the whole stream (though can't remember the whole stream). We discuss linear sketch in some detail and give some example of uses of them. Sketches are generally specific to a particular purpose (meaning sketches are designed to answer specific queries).

9.3 Linear Sketch and Frequency Moments

A linear sketch interprets the stream as defining the frequency vector, as shown in Figure 29.

Often we are interested in functions of the frequency vector from a stream. Given $\mathcal{S} = < a_1, a_2, a_3, \ldots, a_m >$ with $a_i \in [n]$, let f_i be the frequency of i in $\mathcal{S}$ and $\mathbf{F} = \{f_1, f_2, \ldots, f_n\}$ be the frequency vector. Some commonly used functions of the frequency vectors are:

- $F_0 := \sum_{i=1}^{n} f_i^0$ ▷ number of distinct elements
- $F_1 := \sum_{i=1}^{n} f_i$ ▷ length of stream, m

- $F_2 := \sum_{i=1}^{n} f_i^2$ ▷ second frequency moment

Having established the importance of the frequency vector of a data stream, we now attempt to estimate the frequency vector. Recall that storing the exact frequency of items in a data stream is not possible as the stream may consist of possibly infinite and distinct items whereas we can only use a very small and limited amount of memory.

9.4 Count-Min Sketch

PROBLEM 10. *Given a data stream $\mathcal{S} = a_1, a_2, a_3, \ldots, a_m$, where each $a_i \in [n]$, which defines a frequency vector $\mathbf{F} = \{f_1, f_2, \ldots, f_n\}$ where f_i is the frequency of i in the stream, i.e. $f_j = |\{i : a_i = j\}|$, estimate the frequencies f_i for all elements.*

A randomized solution to this problem is the Count-Min sketch. The Count-Min sketch is a simple sketch and has found many applications. It was introduced by Cormode & Muthukrishnan in 2005. The algorithm takes an error bound ϵ and an error probability bound δ and provides an additive approximation guarantee. We begin with a simpler version.

Let $h : [n] \to [1, k]$ be a function chosen uniformly from a 2-universal family of hash functions. A 2-universal family $\mathcal{H}$ of hash functions have that property,

$$Pr_{h \in_R \mathcal{H}}[h(x) = h(y)] = 1/k$$

We keep an array $Count[1 \ldots k]$ of k integers. Consider the following simple algorithm.

Algorithm : Count-Min Sketch (k, ϵ, δ)

```
COUNT ← ZEROS(k)                          ▷ sketch consists of k integers
Pick a random h : [n] ↦ [k] from a 2-universal family H
On input a_i
    COUNT[h(a_i)] ← COUNT[h(a_i)] + 1     ▷ increment count at index h(a_i)

On query j                                ▷ query: F[j] =?
    return COUNT[h(j)]
```

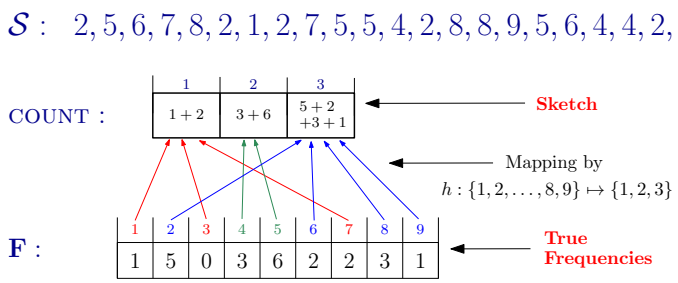


Figure 30: Count-Min Sketch

Note that when $k < n$ (which is typically the case, actually we require that $k \in o(n)$ i.e. $k \ll n$), the algorithm provides an upper bound on actual frequency (since the algorithm returns $Count[h(j)]$, other elements that hash to the same value, i.e. elements i, such that $h(i) = h(j)$ also contribute to the returned value $Count[h(j)]$).

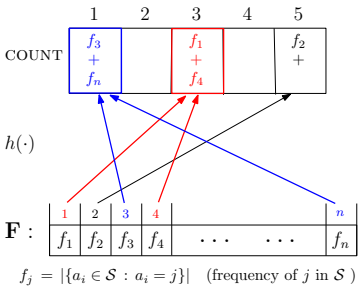


Figure 31: The frequency estimated by the algorithm is an upper bound on the actual frequency.

Let $\tilde{f}_j = Count[h(j)]$ be the estimate provided by the algorithm for query j then from the above reasoning we get that

$$\tilde{f}_j \geq f_j.$$

For $j \in [n]$, we estimate the excess (error), X_j in $\tilde{f}_j$. Clearly, $X_j = \tilde{f}_j - f_j$. Let $1_{h(i)=h(j)}$ be the indicator random variable for the event $h(i) = h(j)$.

$$1_{h(i)=h(j)} = \begin{cases} 1 & \text{if } h(i) = h(j) \\ 0 & \text{otherwise} \end{cases}$$

Note that i makes contribution to $\tilde{f}_j$ iff $h(i) = h(j)$ ($1_{h(i)=h(j)} = 1$) and when it does contribute, its contribution is exactly f_i. We therefore get that

$$X_j = \sum_{i \in [n] \setminus j} f_i \cdot 1_{h(i)=h(j)}.$$

By the *goodness* of h, (h is a 2-universal hash function), we have that $\forall\, i \neq j\ \ \mathbb{P}[h(i) = h(j)] = \frac{1}{k}$. This gives us

$$\mathbb{E}[1_{h(i)=h(j)}] = \mathbb{P}[h(i) = h(j)] = 1/k.$$

We find the expectation of X_j.

$$\mathbb{E}(X_j) = \mathbb{E}(\sum_{i\in[n]\setminus j} f_i \cdot 1_{h(i)=h(j)}) = \sum_{i\in[n]\setminus j} \mathbb{E}(f_i \cdot 1_{h(i)=h(j)}) \qquad \text{Linearity of expectation}$$

$$= \sum_{i\in[n]\setminus j} f_i \cdot \mathbb{E}(1_{h(i)=h(j)}) = \sum_{i\in[n]\setminus j} f_i \cdot \frac{1}{k} \leq \sum_{i\in[n]\setminus j} \|F\|_1 \cdot \frac{1}{k}$$

Since all frequencies are non-negative it is actually the L_1 norm of the frequency vector, that is why we denoted it by the L_1 norm of F.

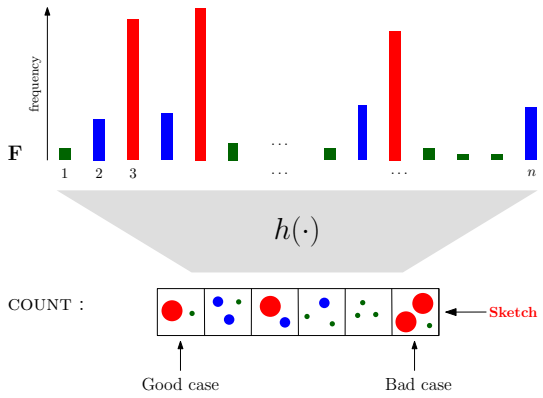


Figure 32: Comparison of good vs. bad case

Recall the Markov's Inequality i.e. If Z be a non-negative random variable, then $\mathbb{P}[Z \geq t \cdot \mathbb{E}(Z)] \leq \frac{1}{t}$

Substitute $k = 2/\epsilon$ (since k, the length of hash table is in our control) and using Markov's inequality we get that

$$\mathbb{P}[X_j \geq \epsilon\|F\|_1] = \mathbb{P}[X_j \geq 2\,\mathbb{E}[X_j]] \leq 1/2$$

Summarizing the analysis of this algorithm we get

- $\tilde{f_j} \geq f_j$
- $\tilde{f_j} \leq f_j + \epsilon\|F\|_1$ with probability at least $1/2$

Hence Algorithm is an $(\epsilon\|F\|_1, \frac{1}{2})$-additive approximation algorithm. Space required by the algorithm is k integers (plus some more for processing etc.) and $k = 2/\epsilon$ is a constant.

We can amplify the probability of success by selecting t independent hash functions $h_1, h_2, \ldots, h_t$ each from a 2-universal family of hash functions and proceed as follows. Each $h_i : [n] \to [1 \ldots k]$

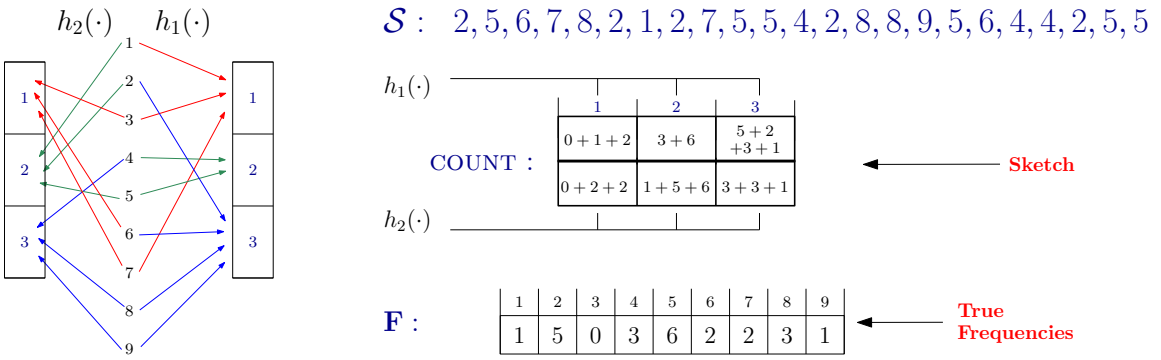


Figure 33: An example of the count-min sketch using two independent hash functions.

Algorithm 13 : Count-Min Sketch (k, ϵ, δ)

COUNT $\leftarrow$ ZEROS$(t \times k)$ ▷ sketch consists of t rows of k integers
Pick t random functions $h_1, \ldots, h_t : [n] \mapsto [k]$ from a 2-universal family
On input a_i
for $r = 1$ to t **do**
    COUNT$[r][h_r(a_i)] \leftarrow$ COUNT$[r][h_r(a_i)] + 1$
    ▷ increment COUNT$[r]$ at index $h_r(a_i)$
On query j ▷ query: $\mathbf{F}[j]$ =?
    return $\min_{1 \le r \le t}$ COUNT$[r][h_r(j)]$

So we keep t estimates instead of 1, and since every estimate is an upper bound, it is clear that we should return the minimum of all estimates (the one which has minimum contribution from other elements).

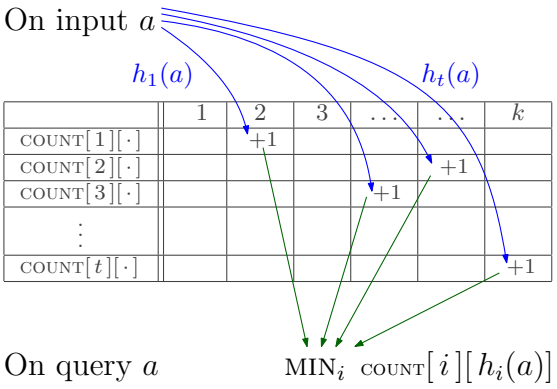


Figure 34: Amplifying probability of the count-min sketch using t independent hash functions

Define X_{jr} to be the contribution of other elements to $Count[r][h(j)]$. We know that if the length of hash table is $k = 2/\epsilon$

$$\mathbb{P}[X_{jr} \geq \epsilon \|F\|_1] \leq 1/2$$

Now if $\tilde{f}_j \geq f_j + \epsilon\|F\|_1$, then for all $1 \leq r \leq t$, we must have that $X_{jr} \geq \epsilon\|F\|_1$, and the probability of this event is (since h_r's are independent) is bounded as

$$\mathbb{P}[\ \forall\ r\ X_{jr} \geq \epsilon\|F\|_1] \leq (1/2)^t$$

Substitute $t = \log(\frac{1}{\delta})$ (the number of hash functions is in our control) and we get

$$\mathbb{P}[\ \forall\ r\ X_{jr} \geq \epsilon\|F\|_1] \leq (1/2)^{\log 1/\delta} = \delta$$

Summarizing the analysis of this algorithm 2 we get

- $\tilde{f}_j \geq f_j$
- $\tilde{f}_j \leq f_j + \epsilon\|F\|_1$ with probability at least $1 - \delta$

Hence Algorithm 13 is an $(\epsilon\|F\|_1, \delta)$ additive approximation algorithm. Space required by the algorithm is $k \cdot t$ integers (plus some more for processing etc), and $k = \frac{2}{\epsilon}$ and $t = \log(1/\delta)$.